JAVA E ORIENTAÇÃO A OBJETOS

Curso FJ-11



Caelum

SOBRE ESTA APOSTILA

Esta apostila da Caelum visa ensinar de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material. Todos os comentários, críticas e sugestões serão muito bem-vindos.

Essa apostila é constantemente atualizada e disponibilizada no site da Caelum. Sempre consulte o site para novas versões e, ao invés de anexar o PDF para enviar a um amigo, indique o site para que ele possa sempre baixar as últimas versões. Você pode conferir o código de versão da apostila logo no nal do índice.

Baixe sempre a versão mais nova em: www.caelum.com.br/apostilas

Sumário

Capítulo 1

COMO APRENDER JAVA

"Busco um instante feliz que justifique minha existência" -- Fiodór Dostoiévski

1.1 O QUE É REALMENTE IMPORTANTE?

Muitos livros, ao passar dos capítulos, mencionam todos os detalhes da linguagem juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial porque o estudante não consegue distinguir exatamente o que é primordial aprender no início, daquilo que pode ser estudado mais adiante.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o if só aceita argumentos booleanos e todos os detalhes sobre classes internas, realmente não devem se tornar preocupações para aquele cujo objetivo primário é aprender Java. Esse tipo de informação será adquirida com o tempo, e não é necessário no início.

Neste curso, separamos essas informações em quadros especiais, já que são informações extras. Ou então, apenas citamos num exercício e deixamos para o leitor procurar informações se for de seu interesse.

Por fim, falta mencionar algo sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos quando o curso terminar. De qualquer maneira recomendamos aos alunos estudarem em casa, e praticarem bastante código e variações.

O curso

Para aqueles que estão fazendo o curso Java e Orientação a Objetos, recomendamos estudarem em casa aquilo que foi visto durante a aula, tentando resolver os exercícios opcionais e os desafios apresentados.

1.2 SOBRE OS EXERCÍCIOS

Os exercícios do curso variam de práticos até pesquisas na Internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos para incitar a curiosidade do aprendiz na tecnologia.

Existem também, em determinados capítulos, uma série de desafios. Eles focam mais no problema computacional que na linguagem, porém são uma excelente forma de treinar a sintaxe e, principalmente, familiarizar o aluno com a biblioteca padrão Java, além de proporcionar um ganho na velocidade de desenvolvimento.

No capítulo 23 há possibilidades de respostas para os exercícios e desafios.

1.3 TIRANDO DÚVIDAS E INDO ALÉM

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do GUJ (http://www.guj.com.br/), onde sua dúvida será respondida prontamente. O GUJ foi fundado por desenvolvedores da Caelum, e hoje conta com mais de um milhão de mensagens.

Fora isso, sinta-se à vontade para entrar em contato com seu instrutor para tirar todas as dúvidas que surgirem durante o curso.

Se o que você está buscando são livros de apoio, sugerimos conhecer a editora Casa do Código:

http://www.casadocodigo.com.br

A Caelum fornece muitos outros cursos Java, com destaque para o FJ-21 que traz a aplicação do Java na web.

http://www.caelum.com.br/

Há também cursos online que vão te ajudar a ir além, com muita interação com os instrutores, a Alura:

http://www.alura.com.br/

CAPÍTULO 2

O QUE É JAVA

"Computadores são inúteis, eles apenas dão respostas" -- *Picasso*

Chegou a hora de responder as perguntas mais básicas sobre Java. Ao término desse capítulo, você será capaz de:

- responder o que é Java;
- mostrar as vantagens e desvantagens do Java;
- entender bem o conceito de máquina virtual;
- compilar e executar um programa simples.

2.1 JAVA

Entender um pouco da história da plataforma Java é essencial para enxergar os motivos que a levaram ao sucesso.

Quais eram os seus maiores problemas quando programava na década de 1990?

ponteiros? gerenciamento de memória? *organização?* falta de bibliotecas? *ter de reescrever parte do código ao mudar de sistema operacional?* custo financeiro de usar a tecnologia?

A linguagem Java resolve bem esses problemas, que até então apareciam com frequência nas outras linguagens. Alguns desses problemas foram particularmente atacados porque uma das grandes motivações para a criação da plataforma Java era de que essa linguagem fosse usada em pequenos dispositivos, como tvs, videocassetes, aspiradores, liquidificadores e outros. Apesar disso a linguagem teve seu lançamento focado no uso em clientes web (browsers) para rodar pequenas aplicações (**applets**). Hoje em dia esse não é o grande mercado do Java: apesar de ter sido idealizado com um propósito e lançado com outro, o Java ganhou destaque no lado do servidor.

O Java foi criado pela antiga Sun Microsystems e mantida através de um comitê (http://www.jcp.org). Seu site principal era o java.sun.com, e java.com um site mais institucional, voltado ao consumidor de produtos e usuários leigos, não desenvolvedores. Com a compra da Sun pela Oracle em 2009, muitas URLs e nomes tem sido trocados para refletir a marca da Oracle. A página principal do Java é: http://www.oracle.com/technetwork/java/

No Brasil, diversos grupos de usuários se formaram para tentar disseminar o conhecimento da

linguagem. Um deles é o **GUJ** (http://www.guj.com.br), uma comunidade virtual com artigos, tutoriais e fórum para tirar dúvidas, o maior em língua portuguesa com mais de cem mil usuários e 1 milhão de mensagens.

Encorajamos todos os alunos a usar muito os fóruns do mesmo, pois é uma das melhores maneiras para achar soluções para pequenos problemas que acontecem com grande frequência.

2.2 UMA BREVE HISTÓRIA DO JAVA

A Sun criou um time (conhecido como Green Team) para desenvolver inovações tecnológicas em 1992. Esse time foi liderado por James Gosling, considerado o pai do Java. O time voltou com a ideia de criar um interpretador (já era uma máquina virtual, veremos o que é isso mais a frente) para pequenos dispositivos, facilitando a reescrita de software para aparelhos eletrônicos, como vídeo cassete, televisão e aparelhos de TV a cabo.

A ideia não deu certo. Tentaram fechar diversos contratos com grandes fabricantes de eletrônicos, como Panasonic, mas não houve êxito devido ao conflito de interesses e custos. Hoje, sabemos que o Java domina o mercado de aplicações para celulares com mais de 2.5 bilhões de dispositivos compatíveis, porém em 1994 ainda era muito cedo para isso.

Com o advento da web, a Sun percebeu que poderia utilizar a ideia criada em 1992 para rodar pequenas aplicações dentro do browser. A semelhança era que na internet havia uma grande quantidade de sistemas operacionais e browsers, e com isso seria grande vantagem poder programar numa única linguagem, independente da plataforma. Foi aí que o Java 1.0 foi lançado: focado em transformar o browser de apenas um cliente magro (*thin client* ou terminal burro) em uma aplicação que possa também realizar operações avançadas, e não apenas renderizar html.

Os applets deixaram de ser o foco da Sun, e nem a Oracle nunca teve interesse. É curioso notar que a

tecnologia Java nasceu com um objetivo em mente, foi lançado com outro, mas, no final, decolou mesmo no desenvolvimento de aplicações do lado do servidor. Sorte? Há hoje o Java FX, tentando dar força para o Java não só no desktop mas como aplicações ricas na web, mas muitos não acreditam que haja espaço para tal, considerando o destino de tecnologias como Adobe Flex e Microsoft Silverlight.

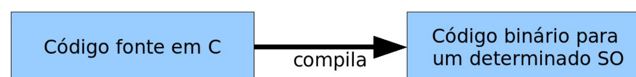
Você pode ler a história da linguagem Java em: http://www.java.com/en/javahistory/

E um vídeo interessante: http://tinyurl.com/histjava

Em 2009 a Oracle comprou a Sun, fortalecendo a marca. A Oracle sempre foi, junto com a IBM, uma das empresas que mais investiram e fizeram negócios através do uso da plataforma Java. Em 2014 surge a versão Java 8 com mudanças interessantes na linguagem.

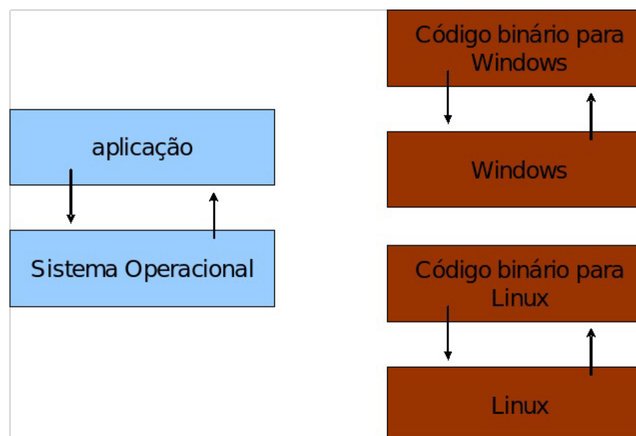
2.3 MÁQUINA VIRTUAL

Em uma linguagem de programação como C e Pascal, temos a seguinte situação quando vamos compilar um programa:



O código fonte é compilado para código de máquina específico de uma plataforma e sistema operacional. Muitas vezes o próprio código fonte é desenvolvido visando uma única plataforma!

Esse código executável (binário) resultante será executado pelo sistema operacional e, por esse motivo, ele deve saber conversar com o sistema operacional em questão.

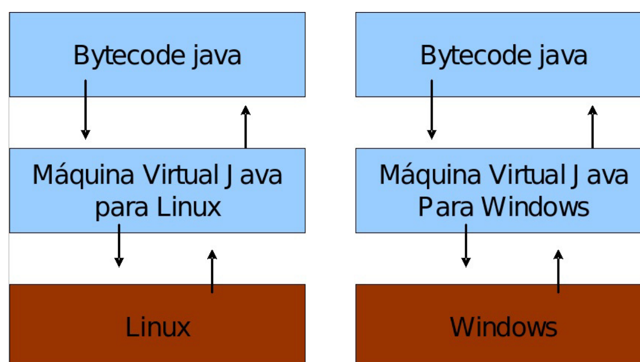


Isto é, temos um código executável para cada sistema operacional. É necessário compilar uma vez para Windows, outra para o Linux, e assim por diante, caso a gente queira que esse nosso software possa ser utilizado em várias plataformas. Esse é o caso de aplicativos como o OpenOffice, Firefox e outros.

Como foi dito anteriormente, na maioria das vezes, a sua aplicação se utiliza das bibliotecas do sistema operacional, como, por exemplo, a de interface gráfica para desenhar as "telas". A biblioteca de interface gráfica do Windows é bem diferente das do Linux: como criar então uma aplicação que rode de forma parecida nos dois sistemas operacionais?

Precisamos reescrever um mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis.

Já o Java utiliza do conceito de **máquina virtual**, onde existe, entre o sistema operacional e a aplicação, uma camada extra responsável por "traduzir" - mas não apenas isso - o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento:



Dessa forma, a maneira com a qual você abre uma janela no Linux ou no Windows é a mesma: você ganha independência de sistema operacional. Ou, melhor ainda, independência de plataforma em geral: não é preciso se preocupar em qual sistema operacional sua aplicação está rodando, nem em que tipo de máquina, configurações, etc.

Repare que uma máquina virtual é um conceito bem mais amplo que o de um interpretador. Como o próprio nome diz, uma máquina virtual é como um "computador de mentira": tem tudo que um computador tem. Em outras palavras, ela é responsável por gerenciar memória, threads, a pilha de execução, etc.

Sua aplicação roda sem nenhum envolvimento com o sistema operacional! Sempre conversando apenas com a **Java Virtual Machine** (JVM).

Essa característica é interessante: como tudo passa pela JVM, ela pode tirar métricas, decidir onde é melhor alocar a memória, entre outros. Uma JVM isola totalmente a aplicação do sistema operacional. Se uma JVM termina abruptamente, só as aplicações que estavam rodando nela irão terminar: isso não afetará outras JVMs que estejam rodando no mesmo computador, nem afetará o sistema operacional.

Essa camada de isolamento também é interessante quando pensamos em um servidor que não pode se sujeitar a rodar código que possa interferir na boa execução de outras aplicações.

Essa camada, a máquina virtual, não entende código java, ela entende um código de máquina específico. Esse código de máquina é gerado por um compilador java, como o **javac**, e é conhecido por "**bytecode**", pois existem menos de 256 códigos de operação dessa linguagem, e cada "opcode" gasta um byte. O compilador Java gera esse bytecode que, diferente das linguagens sem máquina virtual, vai servir para diferentes sistemas operacionais, já que ele vai ser "traduzido" pela JVM.

WRITE ONCE, RUN ANYWHERE

Esse era um slogan que a Sun usava para o Java, já que você não precisa reescrever partes da sua aplicação toda vez que quiser mudar de sistema operacional.

2.4 JAVA LENTO? HOTSPOT E JIT

Hotspot é a tecnologia que a JVM utiliza para detectar *pontos quentes* da sua aplicação: código que é executado muito, provavelmente dentro de um ou mais loops. Quando a JVM julgar necessário, ela vai **compilar** estes códigos para instruções realmente nativas da plataforma, tendo em vista que isso vai provavelmente melhorar a performance da sua aplicação. Esse compilador é o *JIT: Just inTime Compiler*, o compilador que aparece "bem na hora" que você precisa.

Você pode pensar então: porque a JVM não compila tudo antes de executar a aplicação? É que teoricamente compilar dinamicamente, a medida do necessário, pode gerar uma performance melhor. O motivo é simples: imagine um .exe gerado pelo VisualBasic, pelo gcc ou pelo Delphi; ele é estático. Ele já foi otimizado baseado em heurísticas, o compilador pode ter tomado uma decisão não tão boa.

Já a JVM, por estar compilando dinamicamente durante a execução, pode perceber que um determinado código não está com performance adequada e otimizar mais um pouco aquele trecho, ou ainda mudar a estratégia de otimização. É por esse motivo que as JVMs mais recentes em alguns casos chegam a ganhar de códigos C compilados com o GCC 3.x.

Você pode também fazer o curso data dessa apostila na Caelum



Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

2.5 VERSÕES DO JAVA E A CONFUSÃO DO JAVA2

Java 1.0 e 1.1 são as versões muito antigas do Java, mas já traziam bibliotecas importantes como o JDBC e o java.io.

Com o Java 1.2 houve um aumento grande no tamanho da API, e foi nesse momento em que trocaram a nomenclatura de Java para Java2, com o objetivo de diminuir a confusão que havia entre Java e Javascript. Mas lembre-se, não há versão "Java 2.0", o 2 foi incorporado ao nome, tornando-se Java2 1.2.

Depois vieram o Java2 1.3 e 1.4, e o Java 1.5 passou a se chamar Java 5, tanto por uma questão de marketing e porque mudanças significativas na linguagem foram incluídas. É nesse momento que o "2" do nome Java desaparece. Repare que para fins de desenvolvimento, o Java 5 ainda é referido como Java 1.5.

Hoje a última versão disponível do Java é a 8.

2.6 JVM? JRE? JDK? O QUE DEVO BAIXAR?

O que gostaríamos de baixar no site da Oracle?

- JVM = apenas a virtual machine, esse download não existe, ela sempre vem acompanhada.
- JRE = **Java Runtime Environment**, ambiente de execução Java, formado pela JVM e bibliotecas, tudo que você precisa para executar uma aplicação Java. Mas nós precisamos de mais.
- JDK = **Java Development Kit**: Nós, desenvolvedores, faremos o download do JDK do Java SE (Standard Edition). Ele é formado pela JRE somado a ferramentas, como o compilador.

Tanto o JRE e o JDK podem ser baixados do site http://www.oracle.com/technetwork/java/. Para encontrá-los, acesse o link Java SE dentro dos top downloads. Consulte o apêndice de instalação do JDK

para maiores detalhes.

2.7 ONDE USAR E OS OBJETIVOS DO JAVA

No decorrer do curso, você pode achar que o Java tem menor produtividade quando comparada com a linguagem que você está acostumado.

É preciso ficar claro que a premissa do Java não é a de criar sistemas pequenos, onde temos um ou dois desenvolvedores, mais rapidamente que linguagens como php, perl, e outras.

O foco da plataforma é outro: aplicações de *médio a grande porte*, onde o time de desenvolvedores tem *várias pessoas* e sempre pode vir a *mudar* e *crescer*. Não tenha dúvidas que criar a primeira versão de uma aplicação usando Java, mesmo utilizando IDEs e ferramentas poderosas, será mais trabalhoso que muitas linguagens script ou de alta produtividade. Porém, com uma linguagem orientada a objetos e madura como o Java, será extremamente mais fácil e rápido fazer alterações no sistema, desde que você siga as boas práticas e recomendações sobre *design* orientado a objetos.

Além disso, a quantidade enorme de bibliotecas gratuitas para realizar os mais diversos trabalhos (tais como relatórios, gráficos, sistemas de busca, geração de código de barra, manipulação de XML, tocadores de vídeo, manipuladores de texto, persistência transparente, impressão, etc) é um ponto fortíssimo para adoção do Java: você pode criar uma aplicação sofisticada, usando diversos recursos, sem precisar comprar um componente específico, que costuma ser caro. O ecossistema do Java é enorme.

Cada linguagem tem seu espaço e seu melhor uso. O uso do Java é interessante em aplicações que virão a crescer, em que a legibilidade do código é importante, onde temos muita conectividade e se há muitas plataformas (ambientes e sistemas operacionais) heterogêneas (Linux, Unix, OSX e Windows misturados).

Você pode ver isso pela quantidade enorme de ofertas de emprego procurando desenvolvedores Java para trabalhar com sistemas web e aplicações de integração no servidor.

Apesar disto, a Sun empenhou-se em tentar popularizar o uso do Java em aplicações desktop, mesmo com o fraco marketshare do Swing/AWT/SWT em relação às tecnologias concorrentes (em especial Microsoft .NET). A atual tentativa é o Java FX, onde a Oracle tem investido bastante.

2.8 ESPECIFICAÇÃO VERSUS IMPLEMENTAÇÃO

Outro ponto importante: quando falamos de Java Virtual Machine, estamos falando de uma especificação. Ela diz como o bytecode deve ser interpretado pela JVM. Quando fazemos o download no site da Oracle, o que vem junto é a Oracle JVM. Em outras palavras, existem outras JVMs disponíveis, como a JRockit da BEA (também adquirida pela Oracle), a J9 da IBM, entre outras.

Esse é outro ponto interessante para as empresas. Caso não estejam gostando de algum detalhe da JVM da Oracle ou prefiram trabalhar com outra empresa, pagando por suporte, elas podem trocar de JVM com a garantia absoluta de que todo o sistema continuará funcionando. Isso porque toda JVM deve ser certificada pela Oracle, provando a sua compatibilidade. Não há nem necessidade de recompilar nenhuma de suas classes.

Além de independência de hardware e sistema operacional, você tem a independência de *vendor* (fabricante): graças a ideia da JVM ser uma especificação e não um software.

2.9 COMO O FJ-11 ESTÁ ORGANIZADO

Java é uma linguagem simples: existem poucas regras, muito bem definidas.

Porém quebrar o paradigma procedural para mergulhar na orientação a objetos não é simples. Quebrar o paradigma e ganhar fluência com a linguagem e API são os objetivos do FJ-11.

O começo pode ser um pouco frustrante: exemplos simples, controle de fluxo com o `if` , `for` , `while` e criação de pequenos programas que nem ao menos captam dados do teclado. Apesar de isso tudo ser necessário, é só nos 20% finais do curso que utilizaremos bibliotecas para, no final, criarmos um chat entre duas máquinas que transferem Strings por TCP/IP. Neste ponto, teremos tudo que é necessário para entender completamente como a API funciona, quem estende quem, e o porquê.

Depois desse capítulo no qual o Java, a JVM e primeiros conceitos são passados, veremos os comandos básicos do java para controle de fluxo e utilização de variáveis do tipo primitivo. Criaremos classes para testar esse pequeno aprendizado, sem saber exatamente o que é uma classe. Isso dificulta ainda mais a curva de aprendizado, porém cada conceito será introduzido no momento considerado mais apropriado pelos instrutores.

Passamos para o capítulo de orientação a objetos básico, mostrando os problemas do paradigma procedural e a necessidade de algo diferente para resolvê-los. Atributos, métodos, variáveis do tipo referência e outros.

Os capítulos de modificadores de acesso, herança, classes abstratas e interfaces demonstram o conceito fundamental que o curso quer passar: encapsule, exponha o mínimo de suas classes, foque no que elas fazem, no relacionamento entre elas. Com um bom design, a codificação fica fácil e a modificação e expansão do sistema também.

No decorrer desses capítulos, o Eclipse é introduzido de forma natural, evitando-se ao máximo wizards e menus, priorizando mostrar os chamados *code assists* e *quick fixes*. Isso faz com que o Eclipse trabalhe de forma simbiótica com o desenvolvedor, sem se intrometer, sem fazer mágica.

Pacotes, javadoc, jars e `java.lang` apresentam os últimos conceitos fundamentais do Java, dando toda a fundação para, então, passarmos a estudar as principais e mais utilizadas APIs do Java SE.

As APIs estudadas serão `java.util` e `java.io` . Todas elas usam e abusam dos conceitos vistos no decorrer do curso, ajudando a sedimentá-los. Juntamente, temos os conceitos básicos do uso de Threads, e os problemas e perigos da programação concorrente quando dados são compartilhados.

Resumindo: o objetivo do curso é apresentar o Java ao mesmo tempo que os fundamentos da orientação a objetos são introduzidos. Bateremos muito no ponto de dizer que o importante é como as classes se relacionam e qual é o papel de cada uma, e não em como elas realizam as suas obrigações. *Programe voltado à interface, e não à implementação*.

2.10 COMPILANDO O PRIMEIRO PROGRAMA

Vamos para o nosso primeiro código! O programa que imprime uma linha simples.

Para mostrar uma linha, podemos fazer:

```
System.out.println("Minha primeira aplicação Java!");
```

Mas esse código não será aceito pelo compilador java. O Java é uma linguagem bastante burocrática, e precisa de mais do que isso para iniciar uma execução. Veremos os detalhes e os porquês durante os próximos capítulos. O mínimo que precisaríamos escrever é algo como:

```
class MeuPrograma {
    public static void main(String[] args) {
```

```
        System.out.println("Minha primeira aplicação Java!");
    }
}
```

NOTAÇÃO

Todos os códigos apresentados na apostila estão formatados com recursos visuais para auxiliar a leitura e compreensão dos mesmos. Quando for digitar os códigos no computador, trate os códigos como texto simples.

A numeração das linhas **não** faz parte do código e não deve ser digitada; é apenas um recurso didático. O Java é case sensitive: tome cuidado com maiúsculas e minúsculas.

Após digitar o código acima, grave-o como **MeuPrograma.java** em algum diretório. Para compilar, você deve pedir para que o compilador de Java da Oracle, chamado `javac`, gere o bytecode correspondente ao seu código Java.

```
teste@andrade:/home/moreira/java$ javac MeuPrograma.java
teste@andrade:/home/moreira/java$ ls -l
total 8
-rw-r--r-- 1 teste teste 442 2006-09-06 16:41 MeuPrograma.class
-rw-r--r-- 1 teste teste 119 2006-09-06 16:36 MeuPrograma.java
teste@andrade:/home/moreira/java$
```

Depois de compilar, o **bytecode** foi gerado. Quando o sistema operacional listar os arquivos contidos no diretório atual, você poderá ver que um arquivo **.class** foi gerado, com o mesmo nome da sua classe Java.

ASSUSTADO COM O CÓDIGO?

Para quem já tem uma experiência com Java, esse primeiro código é muito simples. Mas, se é seu primeiro código em Java, pode ser um pouco traumatizante. Não deixe de ler o prefácio do curso, que deixará você mais tranquilo em relação a curva de aprendizado da linguagem, conhecendo como o curso está organizado.

PRECISO SEMPRE PROGRAMAR USANDO O NOTEPAD OU SIMILAR?

Não é necessário digitar sempre seu programa em um simples aplicativo como o Notepad. Você pode usar um editor que tenha **syntax highlighting** e outros benefícios.

Mas, no começo, é interessante você usar algo que não possua ferramentas, para que você possa se acostumar com os erros de compilação, sintaxe e outros. Depois do capítulo de polimorfismo e herança sugerimos a utilização do Eclipse (http://www.eclipse.org), a IDE líder no mercado, e gratuita. Existe um capítulo a parte para o uso do Eclipse nesta apostila.

No Linux, recomendamos o uso do gedit, kate e vi. No Windows, você pode usar o Notepad++ ou o TextPad. No Mac, TextMate, Sublime ou mesmo o vi.

2.11 EXECUTANDO SEU PRIMEIRO PROGRAMA

Os procedimentos para executar seu programa são muito simples. O javac é o compilador Java, e o java é o responsável por invocar a máquina virtual para interpretar o seu programa.

```
teste@andrade:/home/moreira/java$ java MeuPrograma
Meu primeiro programa java
teste@andrade:/home/moreira/java$
```

Ao executar, pode ser que a acentuação resultante saia errada devido a algumas configurações que deixamos de fazer. Sem problemas.

2.12 O QUE ACONTECEU?

```
class MeuPrograma {
    public static void main(String[] args) {

        // miolo do programa começa aqui!
        System.out.println("Minha primeira aplicação Java!!");
        // fim do miolo do programa

    }
}
```

O miolo do programa é o que será executado quando chamamos a máquina virtual. Por enquanto, todas as linhas anteriores, onde há a declaração de uma classe e a de um método, não importam para nós nesse momento. Mas devemos saber que toda aplicação Java começa por um ponto de entrada, e este ponto de entrada é o método `main` .

Ainda não sabemos o que é método, mas veremos no capítulo 4. Até lá, não se preocupe com essas declarações. Sempre que um exercício for feito, o código que nos importa sempre estará nesse miolo.

No caso do nosso código, a linha do `System.out.println` faz com que o conteúdo entre aspas seja colocado na tela.

2.13 PARA SABER MAIS: COMO É O BYTECODE?

O `MeuPrograma.class` gerado não é legível por seres humanos (não que seja impossível). Ele está escrito no formato que a virtual machine sabe entender e que foi especificado que ela entendesse.

É como um assembly, escrito para esta máquina em específico. Podemos ler os mnemônicos utilizando a ferramenta javap que acompanha o JDK:

```
javap -c MeuPrograma
```

E a saída:

```
MeuPrograma();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
    Code:
    0:    getstatic    #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    3:   ldc     #3; //String Minha primeira aplicação Java!!
    5:   invokevirtual   #4; //Method java/io/PrintStream.println:
                                (Ljava/lang/String;)V
    8:   return

}
```

É o código acima, que a JVM sabe ler. É o "código de máquina", da máquina virtual.

Um bytecode pode ser revertido para o .java original (com perda de comentários e nomes de

variáveis locais). Caso seu software vá virar um produto de prateleira, é fundamental usar um ofuscador no seu código, que vai embaralhar classes, métodos e um monte de outros recursos (indicamos o http://proguard.sf.net).

2.14 EXERCÍCIOS: MODIFICANDO O HELLO WORLD

1. Altere seu programa para imprimir uma mensagem diferente.
2. Altere seu programa para imprimir duas linhas de texto usando duas linhas de código System.out.
3. Sabendo que os caracteres `\n` representam uma quebra de linhas, imprima duas linhas de texto usando uma única linha de código `System.out`.

2.15 O QUE PODE DAR ERRADO?

Muitos erros podem ocorrer no momento que você rodar seu primeiro código. Vamos ver alguns deles:

Código:

```
class X {
    public static void main (String[] args) {
        System.out.println("Falta ponto e vírgula")
    }
}
```

Erro:

```
X.java:4: ';' expected
        }
        ^
1 error
```

Esse é o erro de compilação mais comum: aquele onde um ponto e vírgula fora esquecido. Repare

que o compilador é explícito em dizer que a linha `4` é a com problemas. Outros erros de compilação podem ocorrer se você escreveu palavras chaves (as que colocamos em negrito) em maiúsculas, esqueceu de abrir e fechar as `{}` , etc.

Durante a execução, outros erros podem aparecer:

- Se você declarar a classe como X, compilá-la e depois tentar usá-la como x minúsculo (java x), o Java te avisa:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
                                        X (wrong name: x)
```

- Se tentar acessar uma classe no diretório ou classpath errado, ou se o nome estiver errado, ocorrerá o seguinte erro:

```
Exception in thread "main" java.lang.NoClassDefFoundError: X
```

- Se esquecer de colocar `static` ou o argumento `String[] args` no método `main` :

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Por exemplo:

```
class X {
    public void main (String[] args) {
        System.out.println("Faltou o static, tente executar!");
    }
}
```

- Se não colocar o método `main` como `public` :

```
Main method not public.
```

Por exemplo:

```
class X {
    static void main (String[] args) {
        System.out.println("Faltou o public");
    }
}
```

2.16 UM POUCO MAIS...

- Procure um colega, ou algum conhecido, que esteja em um projeto Java. Descubra porque Java foi escolhido como tecnologia. O que é importante para esse projeto e o que acabou fazendo do Java a melhor escolha?

2.17 EXERCÍCIOS OPCIONAIS

1. Um arquivo fonte Java deve sempre ter a extensão `.java`, ou o compilador o rejeitará. Além disso, existem algumas outras regras na hora de dar o nome de um arquivo Java. Experimente gravar o código deste capítulo com `OutroNome.java` ou algo similar.

   Compile e verifique o nome do arquivo gerado. Como executar a sua aplicação?

CAPÍTULO 3

VARIÁVEIS PRIMITIVAS E CONTROLE DE FLUXO

"Péssima ideia, a de que não se pode mudar" -- *Montaigne*

Aprenderemos a trabalhar com os seguintes recursos da linguagem Java:

- declaração, atribuição de valores, casting e comparação de variáveis;
- controle de fluxo através de `if e else` ;
- instruções de laço `for e while` , controle de fluxo com break e continue.

3.1 DECLARANDO E USANDO VARIÁVEIS

Dentro de um bloco, podemos declarar variáveis e usá-las. Em Java, toda variável tem um tipo que não pode ser mudado, uma vez que declarado:

```
tipoDaVariavel nomeDaVariavel;
```

Por exemplo, é possível ter uma `idade` que guarda um número inteiro:

```
int idade;
```

Com isso, você declara a variável `idade` , que passa a existir a partir daquela linha. Ela é do tipo `int` , que guarda um número inteiro. A partir daí, você pode usá-la, primeiramente atribuindo valores.

A linha a seguir é a tradução de: **"`idade` deve valer quinze"**.

```
idade = 15;
```

Comentários em Java

Para fazer um comentário em java, você pode usar o `//` para comentar até o final da linha, ou então usar o `/* */` para comentar o que estiver entre eles.

```
 /* comentário daqui,
ate aqui */

// uma linha de comentário sobre a idade
int idade;
```

Além de atribuir, você pode utilizar esse valor. O código a seguir declara novamente a variável `idade` com valor 15 e imprime seu valor na saída padrão através da chamada a `System.out.println`.

```
// declara a idade
int idade;
idade = 15;

// imprime a idade
System.out.println(idade);
```

Por fim, podemos utilizar o valor de uma variável para algum outro propósito, como alterar ou definir uma segunda variável. O código a seguir cria uma variável chamada `idadeNoAnoQueVem` com valor de **idade mais um**.

```
// calcula a idade no ano seguinte
int idadeNoAnoQueVem;
idadeNoAnoQueVem = idade + 1;
```

No mesmo momento que você declara uma variável, também é possível inicializá-la por praticidade:

```
int idade = 15;
```

Você pode usar os operadores +, -, / e * para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente. Além desses operadores básicos, há o operador % (módulo) que nada mais é que o **resto de uma divisão inteira**. Veja alguns exemplos:

```
int quatro = 2 + 2;
int tres = 5 - 2;

int oito = 4 * 2;
int dezesseis = 64 / 4;

int um = 5 % 2; // 5 dividido por 2 dá 2 e tem resto 1;
                // o operador % pega o resto da divisão inteira
```

Como rodar esses códigos?

Você deve colocar esses trechos de código dentro do bloco main que vimos no capítulo anterior. Isto é, isso deve ficar no miolo do programa. Use bastante `System.out.println`, dessa forma você pode ver algum resultado, caso contrário, ao executar a aplicação, nada aparecerá.

Por exemplo, para imprimir a `idade` e a `idadeNoAnoQueVem` podemos escrever o seguinte programa de exemplo:

```
class TestaIdade {

    public static void main(String[] args) {
        // imprime a idade
        int idade = 20;
        System.out.println(idade);

        // gera uma idade no ano seguinte
        int idadeNoAnoQueVem;
        idadeNoAnoQueVem = idade + 1;

        // imprime a idade
        System.out.println(idadeNoAnoQueVem);
    }
}
```

Representar números inteiros é fácil, mas como guardar valores reais, tais como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o `double`, que armazena um número com ponto flutuante (e que também pode armazenar um número inteiro).

```
double pi = 3.14;
double x = 5 * 10;
```

O tipo boolean armazena um valor verdadeiro ou falso, e só: nada de números, palavras ou endereços, como em algumas outras linguagens.

```
boolean verdade = true;
```

`true` e `false` são palavras reservadas do Java. É comum que um `boolean` seja determinado através de uma **expressão booleana**, isto é, um trecho de código que retorna um booleano, como o exemplo:

```
int idade = 30;
boolean menorDeIdade = idade < 18;
```

O tipo char guarda um, e apenas um, caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo char! Por exemplo, ela não pode guardar um código como `''` pois o vazio não é um caractere!

```
char letra = 'a';
```

```
System.out.println(letra);
```

Variáveis do tipo `char` são pouco usadas no dia a dia. Veremos mais a frente o uso das `String` s, que usamos constantemente, porém estas não são definidas por um tipo primitivo.

3.2 TIPOS PRIMITIVOS E VALORES

Esses tipos de variáveis são tipos primitivos do Java: o valor que elas guardam são o real conteúdo da variável. Quando você utilizar o **operador de atribuição** `=` o valor será **copiado**.

```
int i = 5; // i recebe uma cópia do valor 5
int j = i; // j recebe uma cópia do valor de i
i = i + 1; // i vira 6, j continua 5
```

Aqui, `i` fica com o valor de 6. Mas e `j` ? Na segunda linha, `j` está valendo 5. Quando `i` passa a valer 6, será que `j` também muda de valor? Não, pois o valor de um tipo primitivo sempre é copiado.

Apesar da linha 2 fazer `j = i` , a partir desse momento essas variáveis não tem relação nenhuma: o que acontece com uma, não reflete em nada com a outra.

OUTROS TIPOS PRIMITIVOS

Vimos aqui os tipos primitivos que mais aparecem. O Java tem outros, que são o `byte` , `short` , `long` e `float` .

Cada tipo possui características especiais que, para um programador avançado, podem fazer muita diferença.

3.3 EXERCÍCIOS: VARIÁVEIS E TIPOS PRIMITIVOS

1. Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos 15000 reais, em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total no trimestre e a média mensal de gastos.

   Se você estiver fazendo em casa e precisar de ajuda, consulte o capítulo ***Resoluções de Exercícios****.*

3.4 DISCUSSÃO EM AULA: CONVENÇÕES DE CÓDIGO E CÓDIGO LEGÍVEL

Discuta com o instrutor e seus colegas sobre convenções de código Java. Por que existem? Por que são importantes?

Discuta também as vantagens de se escrever código fácil de ler e se evitar comentários em excesso. (Dica: procure por *java code conventions*).

3.5 CASTING E PROMOÇÃO

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo `double`, tentar atribuir ele a uma variável `int` não funciona porque é um código que diz: **"i deve valer d"**, mas não se sabe se `d` realmente é um número inteiro ou não.

```
double d = 3.1415;
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro
int i = d; // não compila
```

Apesar de 5 ser um bom valor para um `int`, o compilador não tem como saber que valor estará dentro desse `double` no momento da execução. Esse valor pode ter sido digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores.

Já no caso a seguir, é o contrário:

```
int i = 5;
double d2 = i;
```

O código acima compila sem problemas, já que um `double` pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo `int` podem ser guardados em uma variável double, então não existem problemas no código acima.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;
int i = (int) d3;
```

O casting foi feito para moldar a variável `d3` como um `int`. O valor de `i` agora é 3.

O mesmo ocorre entre valores `int e long`.

```
long x = 10000;
int i = x; // não compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000;
int i = (int) x;
```

CASOS NÃO TÃO COMUNS DE CASTING E ATRIBUIÇÃO

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados `double` pelo Java. E `float` não pode receber um `double` sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra f, que pode ser maiúscula ou minúscula, indica que aquele literal deve ser tratado como `float` .

Outro caso, que é mais comum:

```
double d = 5;
float f = 3;

float x = f + (float) d;
```

Você precisa do casting porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o `double` .

E, uma observação: no mínimo, o Java armazena o resultado em um `int` , na hora de fazer as contas.

Até casting com variáveis do tipo `char` podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o `boolean` .

Castings possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando a conversão **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente (lembrando que o tipo boolean não pode ser convertido para nenhum outro tipo).

PARA: / DE:	byte	short	char	int	long	float	double
byte	----	*Impl.*	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
short	(byte)	----	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
char	(byte)	(short)	----	*Impl.*	*Impl.*	*Impl.*	*Impl.*
int	(byte)	(short)	(char)	----	*Impl.*	*Impl.*	*Impl.*
long	(byte)	(short)	(char)	(int)	----	*Impl.*	*Impl.*
float	(byte)	(short)	(char)	(int)	(long)	----	*Impl.*
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

Tamanho dos tipos

Na tabela abaixo, estão os tamanhos de cada tipo primitivo do Java.

TIPO	***TAMANHO***
boolean	1 bit
byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

3.6 O IF E O ELSE

A sintaxe do `if` no Java é a seguinte:

```
if (condicaoBooleana) {
    codigo;
}
```

Uma **condição booleana** é qualquer expressão que retorne `true` ou `false` . Para isso, você pode usar os operadores `<` , `>` , `<=` , `>=` e outros. Um exemplo:

```
int idade = 15;
if (idade < 18) {
    System.out.println("Não pode entrar");
}
```

Além disso, você pode usar a cláusula `else` para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
int idade = 15;
if (idade < 18) {
    System.out.println("Não pode entrar");
} else {
    System.out.println("Pode entrar");
}
```

Você pode concatenar expressões booleanas através dos operadores lógicos **"E"** e **"OU"**. O **"E"** é representado pelo `&&` e o **"OU"** é representado pelo `||` .

Um exemplo seria verificar se ele tem menos de 18 anos **e** se ele não é amigo do dono:

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && amigoDoDono == false) {
    System.out.println("Não pode entrar");
}
else {
    System.out.println("Pode entrar");
}
```

Esse código poderia ficar ainda mais legível, utilizando-se o operador de negação, o `!` . Esse operador transforma o resultado de uma expressão booleana de `false` para `true` e vice versa.

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && !amigoDoDono) {
    System.out.println("Não pode entrar");
}
else {
    System.out.println("Pode entrar");
}
```

Repare na linha 3 que o trecho `amigoDoDono == false` virou `!amigoDoDono` . **Eles têm o mesmo valor.**

Para comparar se uma variável tem o **mesmo valor** que outra variável ou valor, utilizamos o operador `==` . Repare que utilizar o operador `=` dentro de um `if` vai retornar um erro de compilação, já que o operador `=` é o de atribuição.

```
int mes = 1;
if (mes == 1) {
    System.out.println("Você deveria estar de férias");
}
```

3.7 O WHILE

O `while` é um comando usado para fazer um **laço (loop)**, isto é, repetir um trecho de código algumas vezes. A ideia é que esse trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int idade = 15;
while (idade < 18) {
    System.out.println(idade);
    idade = idade + 1;
}
```

O trecho dentro do bloco do `while` será executado até o momento em que a condição `idade < 18` passe a ser falsa. E isso ocorrerá exatamente no momento em que `idade == 18` , o que não o fará imprimir `18` .

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i = i + 1;
}
```

Já o `while` acima imprime de 0 a 9.

3.8 O FOR

Outro comando de **loop** extremamente utilizado é o `for` . A ideia é a mesma do `while` : fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas além disso, o `for` isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fiquem mais legíveis, as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento) {
    codigo;
```

```
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1) {
    System.out.println("olá!");
}
```

Repare que esse `for` poderia ser trocado por:

```
int i = 0;
while (i < 10) {
    System.out.println("olá!");
    i = i + 1;
}
```

Porém, o código do `for` indica claramente que a variável `i` serve, em especial, para controlar a quantidade de laços executados. Quando usar o `for`? Quando usar o `while`? Depende do gosto e da ocasião.

PÓS INCREMENTO ++

`i = i + 1` pode realmente ser substituído por `i++` quando isolado, porém, em alguns casos, temos essa instrução envolvida em, por exemplo, uma atribuição:

```
int i = 5;
int x = i++;
```

Qual é o valor de `x`? O de `i`, após essa linha, é 6.

O operador `++`, quando vem após a variável, retorna o valor antigo, e incrementa (pós incremento), fazendo `x` valer 5.

Se você tivesse usado o `++` antes da variável (pré incremento), o resultado seria 6:

```
int i = 5;
int x = ++i; // aqui x valera 6
```

3.9 CONTROLANDO LOOPS

Apesar de termos condições booleanas nos nossos laços, em algum momento, podemos decidir parar o loop por algum motivo especial sem que o resto do laço seja executado.

```
for (int i = x; i < y; i++) {
    if (i % 19 == 0) {
        System.out.println("Achei um número divisível por 19 entre x e y");
        break;
    }
}
```

O código acima vai percorrer os números de x a y e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra chave `break`.

Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave `continue`.

```
for (int i = 0; i < 100; i++) {
    if (i > 50 && i < 60) {
        continue;
    }
    System.out.println(i);
}
```

O código acima não vai imprimir alguns números. (Quais exatamente?)

3.10 ESCOPO DAS VARIÁVEIS

No Java, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro.

```
// aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
```

O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e onde é possível acessá-la.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

```
// aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
while (condicao) {
    // o i ainda vale aqui
    int j = 7;
    // o j passa a existir
}
// aqui o j não existe mais, mas o i continua dentro do escopo
```

No bloco acima, a variável `j` pára de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.

```
EscopoDeVariavel.java:8: cannot find symbol
symbol  : variable j
location: class EscopoDeVariavel
                   System.out.println(j);
                                      ^
1 error
```

O mesmo vale para um if:

```
if (algumBooleano) {
    int i = 5;
}
else {
    int i = 10;
}
System.out.println(i); // cuidado!
```

Aqui a variável `i` não existe fora do `if` e do `else`! Se você declarar a variável antes do `if`, vai haver outro erro de compilação: dentro do `if` e do `else` a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
int i;
if (algumBooleano) {
    i = 5;
}
else {
    i = 10;
}
System.out.println(i);
```

Uma situação parecida pode ocorrer com o `for`:

```
for (int i = 0; i < 10; i++) {
    System.out.println("olá!");
}
System.out.println(i); // cuidado!
```

Neste `for`, a variável `i` morre ao seu término, não podendo ser acessada de fora do `for`, gerando um erro de compilação. Se você realmente quer acessar o contador depois do loop terminar, precisa de algo como:

```
int i;
for (i = 0; i < 10; i++) {
    System.out.println("olá!");
}
System.out.println(i);
```

3.11 UM BLOCO DENTRO DO OUTRO

Um bloco também pode ser declarado dentro de outro. Isto é, um `if` dentro de um `for`, ou um `for` dentro de um `for`, algo como:

```
while (condicao) {
    for (int i = 0; i < 10; i++) {
        // código
    }
}
```

3.12 PARA SABER MAIS

- Vimos apenas os comandos mais usados para controle de fluxo. O Java ainda possui o `do..while` e o `switch`. Pesquise sobre eles e diga quando é interessante usar cada um deles.

- Algumas vezes, temos vários laços encadeados. Podemos utilizar o `break` para quebrar o laço mais interno. Mas, se quisermos quebrar um laço mais externo, teremos de encadear diversos ifs e seu código ficará uma bagunça. O Java possui um artifício chamado **labeled loops**; pesquise sobre eles.

- O que acontece se você tentar dividir um número inteiro por 0? E por 0.0?

- Existe um caminho entre os tipos primitivos que indicam se há a necessidade ou não de casting entre os tipos. Por exemplo, `int -> long -> double` (um `int` pode ser tratado como um `double`, mas não o contrário). Pesquise (ou teste), e posicione os outros tipos primitivos nesse fluxo.

- Além dos operadores de incremento, existem os de decremento, como `--i e i--`. Além desses, você pode usar instruções do tipo `i += x e i -= x`, o que essas instruções fazem? Teste.

3.13 EXERCÍCIOS: FIXAÇÃO DE SINTAXE

Mais exercícios de fixação de sintaxe. Para quem já conhece um pouco de Java, pode ser muito simples; mas recomendamos fortemente que você faça os exercícios para se acostumar com erros de compilação, mensagens do javac, convenção de código, etc...

Apesar de extremamente simples, precisamos praticar a sintaxe que estamos aprendendo. Para cada exercício, crie um novo arquivo com extensão **.java**, e declare aquele estranho cabeçalho, dando nome a uma classe e com um bloco `main` dentro dele:

```
class ExercicioX {
    public static void main(String[] args) {
        // seu exercício vai aqui
    }
}
```

Não copie e cole de um exercício já existente! Aproveite para praticar.

1. Imprima todos os números de 150 a 300.

2. Imprima a soma de 1 até 1000.

3. Imprima todos os múltiplos de 3, entre 1 e 100.

4. Imprima os fatoriais de 1 a 10.

   O fatorial de um número n é n * (n-1) * (n-2) * ... * 1. Lembre-se de utilizar os parênteses.

   O fatorial de 0 é 1

   O fatorial de 1 é (0!) * 1 = 1

   O fatorial de 2 é (1!) * 2 = 2

   O fatorial de 3 é (2!) * 3 = 6

   O fatorial de 4 é (3!) * 4 = 24

   Faça um for que inicie uma variável n (número) como 1 e fatorial (resultado) como 1 e varia n de 1 até 10:

```
int fatorial = 1;
for (int n = 1; n <= 10; n++) {

}
```

5. No código do exercício anterior, aumente a quantidade de números que terão os fatoriais impressos, até 20, 30, 40. Em um determinado momento, além desse cálculo demorar, vai começar a mostrar respostas completamente erradas. Por quê?

   Mude de `int` para `long` para ver alguma mudança.

6. (opcional) Imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc... Para calculá-la, o primeiro elemento vale 0, o segundo vale 1, daí por diante, o n-ésimo elemento vale o (n-1)-ésimo elemento somado ao (n-2)-ésimo elemento (ex: 8 = 5 + 3).

7. (opcional) Escreva um programa que, dada uma variável `x` com algum valor inteiro, temos um novo `x` de acordo com a seguinte regra:

   - se `x` é par, `x = x / 2`
   - se `x` é impar, `x = 3 * x + 1`
   - imprime `x`

- O programa deve parar quando `x` tiver o valor final de 1. Por exemplo, para `x = 13`, a saída será:

  40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

  IMPRIMINDO SEM PULAR LINHA

  Um detalhe importante é que uma quebra de linha é impressa toda vez que chamamos `println`. Para não pular uma linha, usamos o código a seguir:

  ```
  System.out.print(variavel);
  ```

8. (opcional) Imprima a seguinte tabela, usando `for` s encadeados:

```
1
2 4
3 6 9
4 8 12 16
n n*2 n*3 .... n*n
```


3.14 DESAFIOS: FIBONACCI

1. Faça o exercício da série de Fibonacci usando apenas duas variáveis.

CAPÍTULO 4

ORIENTAÇÃO A OBJETOS BÁSICA

"Programação orientada a objetos é uma péssima ideia, que só poderia ter nascido na Califórnia." -- Edsger Dijkstra

Ao término deste capítulo, você será capaz de:

- dizer o que é e para que serve orientação a objetos;
- conceituar classes, atributos e comportamentos;
- entender o significado de variáveis e objetos na memória.

4.1 MOTIVAÇÃO: PROBLEMAS DO PARADIGMA PROCEDURAL

Orientação a objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural.

Consideremos o clássico problema da validação de um CPF. Normalmente, temos um formulário, no qual recebemos essa informação, e depois temos que enviar esses caracteres para uma função que vai validá-lo, como no pseudocódigo abaixo:

```
cpf = formulario->campo_cpf
valida(cpf)
```

Alguém te obriga a sempre validar esse CPF? Você pode, inúmeras vezes, esquecer de chamar esse validador. Mais: considere que você tem 50 formulários e precise validar em todos eles o CPF. Se sua equipe tem 3 programadores trabalhando nesses formulários, quem fica responsável por essa validação? Todos!

A situação pode piorar: na entrada de um novo desenvolvedor, precisaríamos avisá-lo que sempre devemos validar o cpf de um formulário. É nesse momento que nascem aqueles guias de programação para o desenvolvedor que for entrar nesse projeto - às vezes, é um documento enorme. Em outras palavras, **todo** desenvolvedor precisa ficar sabendo de uma quantidade enorme de informações, que, na maioria das vezes, não está realmente relacionado à sua parte no sistema, mas ele **precisa** ler tudo isso, resultando um entrave muito grande!

Outra situação onde ficam claros os problemas da programação procedural, é quando nos encontramos na necessidade de ler o código que foi escrito por outro desenvolvedor e descobrir como ele funciona internamente. Um sistema bem encapsulado não deveria gerar essa necessidade. Em um

sistema grande, simplesmente não temos tempo de ler todo o código existente.

Considerando que você não erre nesse ponto e que sua equipe tenha uma comunicação muito boa (perceba que comunicação excessiva pode ser prejudicial e atrapalhar o andamento), ainda temos outro problema: imagine que, em todo formulário, você também quer que a idade do cliente seja validada - o cliente precisa ter mais de 18 anos. Vamos ter de colocar um `if` ... mas onde? Espalhado por todo seu código... Mesmo que se crie outra função para validar, precisaremos incluir isso nos nossos 50 formulários já existentes. Qual é a chance de esquecermos em um deles? É muito grande.

A responsabilidade de verificar se o cliente tem ou não tem 18 anos ficou espalhada por todo o seu código. Seria interessante poder concentrar essa responsabilidade em um lugar só, para não ter chances de esquecer isso.

Melhor ainda seria se conseguíssemos mudar essa validação e os outros programadores nem precisassem ficar sabendo disso. Em outras palavras, eles criariam formulários e um único programador seria responsável pela validação: os outros nem sabem da existência desse trecho de código. Impossível? Não, o paradigma da orientação a objetos facilita tudo isso.

O problema do paradigma procedural é que não existe uma forma simples de criar conexão forte entre dados e funcionalidades. No paradigma orientado a objetos é muito fácil ter essa conexão através dos recursos da própria linguagem.

Quais as vantagens?

Orientação a objetos vai te ajudar em muito em se organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação, **encapsulando** a lógica de negócios.

Outra enorme vantagem, onde você realmente vai economizar montanhas de código, é o **polimorfismo das referências**, que veremos em um posterior capítulo.

Nos próximos capítulos, conseguiremos enxergar toda essa vantagem, mas, primeiramente é necessário conhecer um pouco mais da sintaxe e da criação de tipos e referências em Java.

4.2 CRIANDO UM TIPO

Considere um programa para um banco, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a conta. Nossa ideia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.

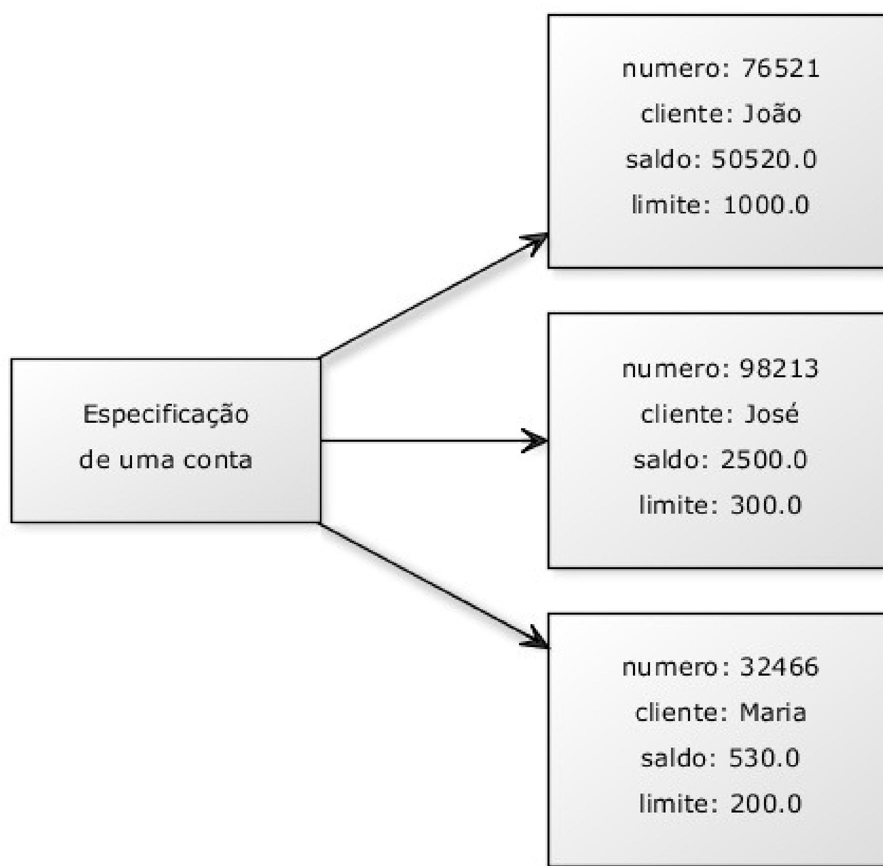
O que toda conta tem e é importante para nós?

- número da conta
- nome do titular da conta
- saldo

O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir à conta"?

- saca uma quantidade x
- deposita uma quantidade x
- imprime o nome do titular da conta
- devolve o saldo atual
- transfere uma quantidade x para uma outra conta y
- devolve o tipo de conta

Com isso, temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não. O que temos ainda é o **projeto**. Antes, precisamos **construir** uma conta, para poder acessar o que ela tem, e pedir a ela que faça algo.



Repare na figura: apesar do papel do lado esquerdo especificar uma Conta, essa especificação é uma Conta? Nós depositamos e sacamos dinheiro desse papel? Não. Utilizamos a especificação da Conta para poder criar instâncias que realmente são contas, onde podemos realizar as operações que criamos.

Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de papel (como à esquerda na figura), são nas instâncias desse projeto que realmente há espaço para armazenar esses valores.

Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**. Ao que podemos construir a partir desse projeto, as contas de verdade, damos o nome de **objetos**.

A palavra **classe** vem da taxonomia da biologia. Todos os seres vivos de uma mesma **classe** biológica têm uma série de **atributos** e **comportamentos** em comum, mas não são iguais, podem variar nos valores desses **atributos** e como realizam esses **comportamentos**.

Homo Sapiens define um grupo de seres que possuem características em comum, porém a definição (a ideia, o conceito) de um **Homo Sapiens** é um ser humano? Não. Tudo está especificado na **classe** Homo Sapiens, mas se quisermos mandar alguém correr, comer, pular, precisaremos de uma instância de **Homo Sapiens**, ou então de um **objeto** do tipo **Homo Sapiens**.

Um outro exemplo: uma receita de bolo. A pergunta é certeira: você come uma receita de bolo? Não.

Precisamos **instanciá-la**, criar um **objeto** bolo a partir dessa especificação (a classe) para utilizá-la. Podemos criar centenas de bolos a partir dessa classe (a receita, no caso), eles podem ser bem semelhantes, alguns até idênticos, mas são **objetos** diferentes.

Podemos fazer milhares de analogias semelhantes. A planta de uma casa é uma casa? Definitivamente não. Não podemos morar dentro da planta de uma casa, nem podemos abrir sua porta ou pintar suas paredes. Precisamos, antes, construir instâncias a partir dessa planta. Essas instâncias, sim, podemos pintar, decorar ou morar dentro.

Pode parecer óbvio, mas a dificuldade inicial do paradigma da orientação a objetos é justo saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar, obviamente de forma errada, essas duas palavras como sinônimos.

4.3 UMA CLASSE EM JAVA

Vamos começar apenas com o que uma `Conta` tem, e não com o que ela faz (veremos logo em seguida).

Um tipo desses, como o especificado de `Conta` acima, pode ser facilmente traduzido para Java:

```
class Conta {
    int numero;
    String titular;
    double saldo;

    // ..
}
```

STRING

String é uma classe em Java. Ela guarda uma cadeia de caracteres, uma frase completa. Como estamos ainda aprendendo o que é uma classe, entenderemos com detalhes a classe `String` apenas em capítulos posteriores.

Por enquanto, declaramos o que toda conta deve ter. Estes são os **atributos** que toda conta, quando criada, vai ter. Repare que essas variáveis foram declaradas fora de um bloco, diferente do que fazíamos quando tinha aquele main. Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.

4.4 CRIANDO E USANDO UM OBJETO

Já temos uma classe em Java que especifica o que todo objeto dessa classe deve ter. Mas como usá-

la? Além dessa classe, ainda teremos o **Programa.java** e a partir dele é que vamos utilizar a classe `Conta`.

Para criar (construir, instanciar) uma `Conta`, basta usar a palavra chave `new`. Devemos utilizar também os parênteses, que descobriremos o que fazem exatamente em um capítulo posterior:

```
class Programa {
    public static void main(String[] args) {
        new Conta();
    }
}
```

Bem, o código acima cria um objeto do tipo `Conta`, mas como acessar esse objeto que foi criado? Precisamos ter alguma forma de nos referenciarmos a esse objeto. Precisamos de uma variável:

```
class Programa {
    public static void main(String[] args) {
        Conta minhaConta;
        minhaConta = new Conta();
    }
}
```

Pode parecer estranho escrevermos duas vezes `Conta`: uma vez na declaração da variável e outra vez no uso do `new`. Mas há um motivo, que em breve entenderemos.

Através da variável `minhaConta`, podemos acessar o objeto recém criado para alterar seu `titular`, seu `saldo`, etc:

```
class Programa {
    public static void main(String[] args) {
        Conta minhaConta;
        minhaConta = new Conta();

        minhaConta.titular = "Duke";
        minhaConta.saldo = 1000.0;

        System.out.println("Saldo atual: " + minhaConta.saldo);
    }
}
```

É importante fixar que o *ponto* foi utilizado para acessar algo em `minhaConta`. A `minhaConta` pertence ao Duke, e tem saldo de mil reais.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

4.5 MÉTODOS

Dentro da classe, também declararemos o que cada conta faz e como isto é feito - os comportamentos que cada classe tem, isto é, o que ela faz. Por exemplo, de que maneira que uma Conta saca dinheiro? Especificaremos isso dentro da própria classe `Conta`, e não em um local desatrelado das informações da própria Conta. É por isso que essas "funções" são chamadas de **métodos**. Pois é a maneira de fazer uma operação com um objeto.

Queremos criar um método que **saca** uma determinada **quantidade** e não devolve **nenhuma informação** para quem acionar esse método:

```
class Conta {
    double salario;
    // ... outros atributos ...

    void saca(double quantidade) {
        double novoSaldo = this.saldo - quantidade;
        this.saldo = novoSaldo;
    }
}
```

A palavra chave `void` diz que, quando você pedir para a conta sacar uma quantia, nenhuma informação será enviada de volta a quem pediu.

Quando alguém pedir para sacar, ele também vai dizer quanto quer sacar. Por isso precisamos declarar o método com algo dentro dos parênteses - o que vai aí dentro é chamado de **argumento** do método (ou **parâmetro**). Essa variável é uma variável comum, chamada também de temporária ou local, pois, ao final da execução desse método, ela deixa de existir.

Dentro do método, estamos declarando uma nova variável. Essa variável, assim como o argumento, vai morrer no fim do método, pois este é seu escopo. No momento que vamos acessar nosso atributo,

usamos a palavra chave `this` para mostrar que esse é um atributo, e não uma simples variável. (veremos depois que é opcional)

Repare que, nesse caso, a conta poderia estourar um limite fixado pelo banco. Mais para frente, evitaremos essa situação, e de uma maneira muito elegante.

Da mesma forma, temos o método para depositar alguma quantia:

```
class Conta {
    // ... outros atributos e métodos ...

    void deposita(double quantidade) {
        this.saldo += quantidade;
    }
}
```

Observe que não usamos uma variável auxiliar e, além disso, usamos a abreviação `+=` para deixar o método bem simples. O `+=` soma quantidade ao valor antigo do saldo e guarda no próprio saldo, o valor resultante.

Para mandar uma mensagem ao objeto e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é **invocação de método**.

O código a seguir saca dinheiro e depois deposita outra quantia na nossa conta:

```
class TestaAlgunsMetodos {
    public static void main(String[] args) {
        // criando a conta
        Conta minhaConta;
        minhaConta = new Conta();

        // alterando os valores de minhaConta
        minhaConta.titular = "Duke";
        minhaConta.saldo = 1000;

        // saca 200 reais
        minhaConta.saca(200);

        // deposita 500 reais
        minhaConta.deposita(500);
        System.out.println(minhaConta.saldo);
    }
}
```

Uma vez que seu saldo inicial é 1000 reais, se sacarmos 200 reais, depositarmos 500 reais e imprimirmos o valor do saldo, o que será impresso?

4.6 MÉTODOS COM RETORNO

Um método sempre tem que definir o que retorna, nem que defina que não há retorno, como nos exemplos anteriores onde estávamos usando o `void`.

Um método pode retornar um valor para o código que o chamou. No caso do nosso método `saca`, podemos devolver um valor booleano indicando se a operação foi bem sucedida.

```
class Conta {
    // ... outros métodos e atributos ...

    boolean saca(double valor) {
        if (this.saldo < valor) {
            return false;
        }
        else {
            this.saldo = this.saldo - valor;
            return true;
        }
    }
}
```

A declaração do método mudou! O método `saca` não tem `void` na frente. Isto quer dizer que, quando é acessado, ele devolve algum tipo de informação. No caso, um `boolean`. A palavra chave `return` indica que o método vai terminar ali, retornando tal informação.

Conta
+numero: int +saldo: double +limite: double +nome: String
+saca(valor: double): boolean +deposita(valor: double)

Exemplo de uso:

```
minhaConta.saldo = 1000;
boolean consegui = minhaConta.saca(2000);
if (consegui) {
    System.out.println("Consegui sacar");
} else {
    System.out.println("Não consegui sacar");
}
```

Ou então, posso eliminar a variável temporária, se desejado:

```
minhaConta.saldo = 1000;
if (minhaConta.saca(2000)) {
    System.out.println("Consegui sacar");
} else {
    System.out.println("Não consegui sacar");
}
```

Mais adiante, veremos que algumas vezes é mais interessante lançar uma exceção (*exception*) nesses casos.

Meu programa pode manter na memória não apenas uma conta, como mais de uma:

```
class TestaDuasContas {
    public static void main(String[] args) {

        Conta minhaConta;
        minhaConta = new Conta();
        minhaConta.saldo = 1000;

        Conta meuSonho;
        meuSonho = new Conta();
        meuSonho.saldo = 1500000;
    }
}
```

4.7 OBJETOS SÃO ACESSADOS POR REFERÊNCIAS

Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**.

É por esse motivo que, diferente dos *tipos primitivos* como `int` e `long`, precisamos dar `new` depois de declarada a variável:

```
    public static void main(String[] args) {
        Conta c1;
        c1 = new Conta();

        Conta c2;
        c2 = new Conta();
    }
```

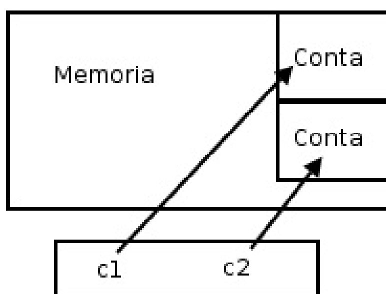
O correto aqui, é dizer que `c1` se refere a um objeto. **Não é correto** dizer que `c1` é um objeto, pois `c1` é uma variável referência, apesar de, depois de um tempo, os programadores Java falarem "Tenho um **objeto c** do tipo **Conta**", mas apenas para encurtar a frase "Tenho uma **referência c** a um **objeto** do tipo **Conta**".

Basta lembrar que, em Java, **uma variável nunca é um objeto**. Não há, no Java, uma maneira de criarmos o que é conhecido como "*objeto pilha*" ou "*objeto local*", pois todo objeto em Java, sem exceção, é acessado por uma variável referência.

Esse código nos deixa na seguinte situação:

```
Conta c1;
c1 = new Conta();

Conta c2;
c2 = new Conta();
```



Internamente, `c1` e `c2` vão guardar um número que identifica em que posição da memória aquela `Conta` se encontra. Dessa maneira, ao utilizarmos o "." para navegar, o Java vai acessar a `Conta` que se encontra naquela posição de memória, e não uma outra.

Para quem conhece, é parecido com um ponteiro, porém você não pode manipulá-lo como um número e nem utilizá-lo para aritmética, ela é tipada.

Um outro exemplo:

```
class TestaReferencias {
    public static void main(String[] args) {
        Conta c1 = new Conta();
        c1.deposita(100);

        Conta c2 = c1;  // linha importante!
        c2.deposita(200);

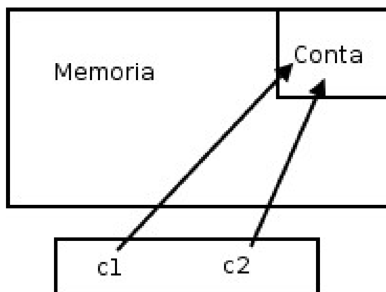
        System.out.println(c1.saldo);
         System.out.println(c2.saldo);
    }
}
```

Qual é o resultado do código acima? O que aparece ao rodar?

O que acontece aqui? O operador `=` copia o valor de uma variável. Mas qual é o valor da variável `c1` ? É o objeto? Não. Na verdade, o valor guardado é a referência (**endereço**) de onde o objeto se encontra na memória principal.

Na memória, o que acontece nesse caso:

```
Conta c1 = new Conta();
Conta c2 = c1;
```



Quando fizemos `c2 = c1` , `c2` passa a fazer referência para o mesmo objeto que `c1` referencia nesse instante.

Então, nesse código em específico, quando utilizamos `c1` ou `c2` estamos nos referindo exatamente ao **mesmo** objeto! Elas são duas referências distintas, porém apontam para o **mesmo** objeto! Compará-las com " == " vai nos retornar `true` , pois o valor que elas carregam é o mesmo!

Outra forma de perceber, é que demos apenas um `new` , então só pode haver um objeto `Conta` na memória.

Atenção: não estamos discutindo aqui a utilidade de fazer uma referência apontar pro mesmo objeto que outra. Essa utilidade ficará mais clara quando passarmos variáveis do tipo referência como argumento para métodos.

NEW

O que exatamente faz o `new` ?

O `new` executa uma série de tarefas, que veremos mais adiante.

Mas, para melhor entender as referências no Java, saiba que o `new` , depois de alocar a memória para esse objeto, devolve uma "flecha", isto é, um valor de referência. Quando você atribui isso a uma variável, essa variável passa a se referir para esse mesmo objeto.

Podemos então ver outra situação:

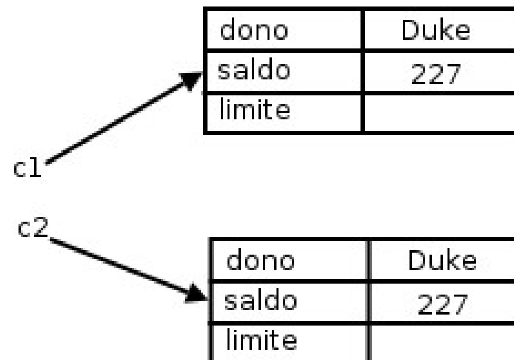
```
public static void main(String[] args) {
    Conta c1 = new Conta();
    c1.titular = "Duke";
    c1.saldo = 227;

    Conta c2 = new Conta();
    c2.titular = "Duke";
    c2.saldo = 227;

    if (c1 == c2) {
        System.out.println("Contas iguais");
    }
}
```

O operador `==` compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, elas estão em espaços diferentes da memória, o que faz o teste no `if` valer `false` . As contas podem ser equivalentes no nosso critério de igualdade, porém elas não são o mesmo objeto. Quando se trata de objetos, pode ficar mais fácil pensar que o `==` compara se os objetos (referências, na

verdade) são o mesmo, e não se são iguais.



Para saber se dois objetos têm o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.

Já conhece os cursos online Alura?

alura

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra, Business, entre outras, com um plano que dá acesso a todos os cursos. Ex-estudante da Caelum tem 10% de desconto neste link!

Conheça os cursos online Alura.

4.8 O MÉTODO TRANSFERE()

E se quisermos ter um método que transfere dinheiro entre duas contas? Podemos ficar tentados a criar um método que recebe dois parâmetros: `conta1` e `conta2` do tipo `Conta` . Mas cuidado: assim estamos pensando de maneira procedural.

A ideia é que, quando chamarmos o método `transfere` , já teremos um objeto do tipo `Conta` (o `this`), portanto o método recebe apenas **um** parâmetro do tipo `Conta` , a Conta destino (além do `valor`):

```
class Conta {

    // atributos e métodos...

    void transfere(Conta destino, double valor) {
```

```
            this.saldo = this.saldo - valor;
            destino.saldo = destino.saldo + valor;
        }
    }
```

Conta
+numero: int +saldo: double +limite: double +nome: String
+saca(valor: double): boolean +deposita(valor: double) +transfere(destino: Conta, valor: double)

Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos `deposita` e `saca` já existentes para fazer essa tarefa:

```
    class Conta {

        // atributos e métodos...

        boolean transfere(Conta destino, double valor) {
            boolean retirou = this.saca(valor);
            if (retirou == false) {
                // não deu pra sacar!
                return false;
            }
            else {
                destino.deposita(valor);
                return true;
            }
        }
    }
```

Conta
+numero: int +saldo: double +limite: double +nome: String
+saca(valor: double): boolean +deposita(valor: double) +transfere(destino: Conta, valor: double): boolean

Quando passamos uma `Conta` como argumento, o que será que acontece na memória? Será que o objeto é *clonado*?

No Java, a passagem de parâmetro funciona como uma simples atribuição como no uso do "=". Então, esse parâmetro vai copiar o valor da variável do tipo `Conta` que for passado como argumento. E qual é o valor de uma variável dessas? Seu valor é um endereço, uma referência, nunca um objeto. Por

isso não há cópia de objetos aqui.

Esse último código poderia ser escrito com uma sintaxe muito mais sucinta. Como?

TRANSFERE PARA

Perceba que o nome deste método poderia ser `transferePara` ao invés de só `transfere`. A chamada do método fica muito mais natural, é possível ler a frase em português que ela tem um sentido:

```
conta1.transferePara(conta2, 50);
```

A leitura deste código seria " `Conta1` transfere para `conta2` 50 reais".

4.9 CONTINUANDO COM ATRIBUTOS

As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão. No caso numérico, valem `0`, no caso de `boolean`, valem `false`.

Você também pode dar **valores default**, como segue:

```
class Conta {
    int numero = 1234;
    String titular = "Duke";
    double saldo = 1000.0;
}
```

Nesse caso, quando você criar uma conta, seus atributos já estão "populados" com esses valores colocados.

Imagine que comecemos a aumentar nossa classe `Conta` e adicionar `nome, sobrenome e cpf` do titular da conta. Começaríamos a ter muitos atributos... e, se você pensar direito, uma `Conta` não tem `nome`, nem `sobrenome` nem `cpf`, quem tem esses atributos é um `Cliente`. Então podemos criar uma nova classe e fazer uma composição

Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe `Cliente`:

```
class Cliente {
    String nome;
    String sobrenome;
    String cpf;
}

class Conta {
    int numero;
    double saldo;
```

```
    Cliente titular;
    // ..
}
```

E dentro do `main` da classe de teste:

```
class Teste {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        Cliente c = new Cliente();
        minhaConta.titular = c;
        // ...
    }
}
```

Aqui, simplesmente houve uma atribuição. O valor da variável `c` é copiado para o atributo titular do objeto ao qual `minhaConta` se refere. Em outras palavras, `minhaConta` tem uma referência ao mesmo `Cliente` que `c` se refere, e pode ser acessado através de `minhaConta.titular`.

Você pode realmente navegar sobre toda essa estrutura de informação, sempre usando o ponto:

```
Cliente clienteDaMinhaConta = minhaConta.titular;
clienteDaMinhaConta.nome = "Duke";
```

Ou ainda, pode fazer isso de uma forma mais direta e até mais elegante:

```
minhaConta.titular.nome = "Duke";
```

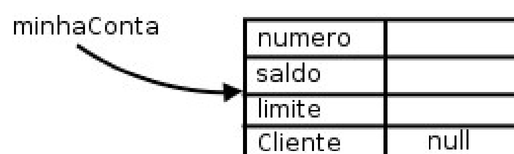
Um sistema orientado a objetos é um grande conjunto de classes que vai se comunicar, delegando responsabilidades para quem for mais apto a realizar determinada tarefa. A classe `Banco` usa a classe `Conta` que usa a classe `Cliente`, que usa a classe `Endereco`. Dizemos que esses objetos colaboram, trocando mensagens entre si. Por isso acabamos tendo muitas classes em nosso sistema, e elas costumam ter um tamanho relativamente curto.

Mas, e se dentro do meu código eu não desse `new` em `Cliente` e tentasse acessá-lo diretamente?

```
class Teste {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();

        minhaConta.titular.nome = "Manoel";
        // ...
    }
}
```

Quando damos `new` em um objeto, ele o inicializa com seus valores default, 0 para números, `false` para `boolean` e `null` para referências. `null` é uma palavra chave em java, que indica uma referência para nenhum objeto.



Se, em algum caso, você tentar acessar um atributo ou método de alguém que está se referenciando para `null`, você receberá um erro durante a execução (`NullPointerException` , que veremos mais à frente). Da para perceber, então, que o `new` não traz um efeito cascata, a menos que você dê um valor default (ou use construtores, que também veremos mais a frente):

```
class Conta {
    int numero;
    double saldo;
    Cliente titular = new Cliente();    // quando chamarem new Conta,
                                        //havera um new Cliente para ele.
}
```

Com esse código, toda nova `Conta` criada já terá um novo `Cliente` associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma `Conta` . Qual alternativa você deve usar? Depende do caso: para toda nova `Conta` você precisa de um novo `Cliente` ? É essa pergunta que deve ser respondida. Nesse nosso caso a resposta é não, mas depende do nosso problema.

Atenção: para quem não está acostumado com referências, pode ser bastante confuso pensar sempre em como os objetos estão na memória para poder tirar as conclusões de o que ocorrerá ao executar determinado código, por mais simples que ele seja. Com tempo, você adquire a habilidade de rapidamente saber o efeito de atrelar as referências, sem ter de gastar muito tempo para isso. É importante, nesse começo, você estar sempre pensando no estado da memória. E realmente lembrar que, no Java *"uma variável nunca carrega um objeto, e sim uma referência para ele"* facilita muito.

4.10 PARA SABER MAIS: UMA FÁBRICA DE CARROS

Além do `Banco` que estamos criando, vamos ver como ficariam certas classes relacionadas a uma fábrica de carros. Vamos criar uma classe `Carro` , com certos atributos, que descrevem suas características, e com certos métodos, que descrevem seu comportamento.

```
class Carro {
    String cor;
    String modelo;
    double velocidadeAtual;
    double velocidadeMaxima;

    //liga o carro
    void liga() {
        System.out.println("O carro está ligado");
    }

    //acelera uma certa quantidade
    void acelera(double quantidade) {
        double velocidadeNova = this.velocidadeAtual + quantidade;
        this.velocidadeAtual = velocidadeNova;
    }

    //devolve a marcha do carro
    int pegaMarcha() {
        if (this.velocidadeAtual < 0) {
            return -1;
```

```
        }
        if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
            return 1;
        }
        if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80) {
            return 2;
        }
        return 3;
    }
}
```

Vamos testar nosso `Carro` em um novo programa:

```
class TestaCarro {
    public static void main(String[] args) {
        Carro meuCarro;
        meuCarro = new Carro();
        meuCarro.cor = "Verde";
        meuCarro.modelo = "Fusca";
        meuCarro.velocidadeAtual = 0;
        meuCarro.velocidadeMaxima = 80;

        // liga o carro
        meuCarro.liga();

        // acelera o carro
        meuCarro.acelera(20);
        System.out.println(meuCarro.velocidadeAtual);
    }
}
```

Nosso carro pode conter também um `Motor` :

```
class Motor {
    int potencia;
    String tipo;
}

class Carro {
    String cor;
    String modelo;
    double velocidadeAtual;
    double velocidadeMaxima;
    Motor motor;

    // ..
}
```

Podemos, criar diversos Carros e mexer com seus atributos e métodos, assim como fizemos no exemplo do `Banco` .

Você pode também fazer o curso data dessa apostila na Caelum



Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

4.11 UM POUCO MAIS...

- Quando declaramos uma classe, um método ou um atributo, podemos dar o nome que quisermos, seguindo uma regra. Por exemplo, o nome de um método não pode começar com um número. Pesquise sobre essas regras.
- Como você pode ter reparado, sempre damos nomes às variáveis com letras minúsculas. É que existem **convenções de código**, dadas pela Oracle, para facilitar a legibilidade do código entre programadores. Essa convenção é *muito seguida*. Leia sobre ela pesquisando por "*java code conventions*".
- É necessário usar a palavra chave `this` quando for acessar um atributo? Para que, então, utilizá-la?
- Existe um padrão para representar suas classes em diagramas, que é amplamente utilizado, chamado **UML**. Pesquise sobre ele.

4.12 EXERCÍCIOS: ORIENTAÇÃO A OBJETOS

O modelo da conta a seguir será utilizado para os exercícios dos próximos capítulos.

O objetivo aqui é criar um sistema para gerenciar as contas de um `Banco` . **Os exercícios desse capítulo são extremamente importantes**.

1. Modele uma conta. A ideia aqui é apenas modelar, isto é, só identifique que informações são importantes. Desenhe no papel tudo o que uma `Conta` tem e tudo o que ela faz. Ela deve ter o nome do titular (`String`), o número (`int`), a agência (`String`), o saldo (`double`) e uma data de abertura (`String`). Além disso, ela deve fazer as seguintes ações: saca, para retirar um valor do saldo; deposita, para adicionar um valor ao saldo; calculaRendimento, para devolver o rendimento

mensal dessa conta.

2. Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o `main`. Você deve criar a classe da conta com o nome `Conta`, mas pode nomear como quiser a classe de testes, por exemplo pode chamá-la `TestaConta`, contudo, ela deve necessariamente possuir o método `main`.

   A classe Conta deve conter, além dos atributos mencionados anteriormente, pelo menos os seguintes métodos:

   - `saca` que recebe um `valor` como parâmetro e retira esse valor do saldo da conta
   - `deposita` que recebe um `valor` como parâmetro e adiciona esse valor ao saldo da conta
   - `calculaRendimento` que não recebe parâmetro algum e devolve o valor do saldo multiplicado por 0.1

     Lembre-se de seguir a convenção java, isso é importantíssimo. Isto é, preste atenção nas maiúsculas e minúsculas, seguindo o seguinte exemplo: `nomeDeAtributo`, `nomeDeMetodo`, `nomeDeVariavel`, `NomeDeClasse`, etc...

> **TODAS AS CLASSES NO MESMO ARQUIVO?**
>
> Você até pode colocar todas as classes no mesmo arquivo e apenas compilar esse arquivo. Ele vai gerar um `.class` para cada classe presente nele.
>
> Porém, por uma questão de organização, é boa prática criar um arquivo `.java` para cada classe. Em capítulos posteriores, veremos também determinados casos nos quais você será **obrigado** a declarar cada classe em um arquivo separado.
>
> Essa separação não é importante nesse momento do aprendizado, mas se quiser ir praticando sem ter que compilar classe por classe, você pode dizer para o `javac` compilar todos os arquivos java de uma vez:
>
> ```
> javac *.java
> ```

3. Na classe Conta, crie um método `recuperaDadosParaImpressao()`, que não recebe parâmetro mas devolve o texto com todas as informações da nossa conta para efetuarmos a impressão.

   Dessa maneira, você não precisa ficar copiando e colando um monte de `System.out.println()` para cada mudança e teste que fizer com cada um de seus funcionários, você simplesmente vai fazer:

```java
Conta c1 = new Conta();
```

```
// brincadeiras com c1....
System.out.println(c1.recuperaDadosParaImpressao());
```

Veremos mais a frente o método `toString`, que é uma solução muito mais elegante para mostrar a representação de um objeto como `String`, além de não jogar tudo pro `System.out` (só se você desejar).

4. Na classe de teste dentro do bloco main, construa duas contas com o `new` e compare-os com o `==`. E se eles tiverem os mesmos atributos? Para isso você vai precisar criar outra referência:

```
Conta c1 = new Conta();
c1.titular = "Danilo";
c1.saldo = 100;

Conta c2 = new Conta();
c2.titular = "Danilo";
c2.saldo = 100;

if (c1 == c2) {
    System.out.println("iguais");
} else {
    System.out.println("diferentes");
}
```

5. Agora, crie duas referências para a **mesma** conta, compare-os com o `==`. Tire suas conclusões. Para criar duas referências pra mesma conta:

```
Conta c1 = new Conta():
c1.titular = "Hugo";
c1.saldo = 100;

c2 = c1;
```

O que acontece com o `if` do exercício anterior?

6. (opcional) Em vez de utilizar uma `String` para representar a data, crie uma outra classe, chamada `Data`. Ela possui 3 campos `int`, para dia, mês e ano. Faça com que sua conta passe a usá-la. (é parecido com o último exemplo da explicação, em que a `Conta` passou a ter referência para um `Cliente`).

```
class Conta {
    Data dataDeAbertura; // qual é o valor default aqui?
    // seus outros atributos e métodos
}

class Data {
    int dia;
    int mes;
    int ano;
}
```

Modifique sua classe `TestaConta` para que você crie uma `Data` e atribua ela a `Conta`:

```
Conta c1 = new Conta();
//...
```

```
    Data data = new Data(); // ligação!
    c1.dataDeAbertura = data;
```

Faça o desenho do estado da memória quando criarmos um `Conta` .

7. (opcional) Modifique seu método `recuperaDadosParaImpressao` para que ele devolva o valor da `dataDeAbertura` daquela `Conta` :

```
class Conta {

    // seus outros atributos e métodos
    Data dataDeAbertura;

    String recuperaDadosParaImpressao() {
        String dados = "\nTitular: " + this.titular;
        // imprimir aqui os outros atributos...

        dados += "\nDia: " + this.dataDeAbertura.dia;
        dados += "\nMês: " + this.dataDeAbertura.mes;
        dados += "\nAno: " + this.dataDeAbertura.ano;
        return dados;
    }
}
```

Teste-o. O que acontece se chamarmos o método `recuperaDadosParaImpressao` antes de atribuirmos uma data para esta `Conta` ?

8. (opcional) O que acontece se você tentar acessar um atributo diretamente na classe? Como, por exemplo:

```
    Conta.saldo = 1234;
```

Esse código faz sentido? E este:

```
    Conta.calculaRendimento();
```

Faz sentido perguntar para o esquema da Conta seu valor anual?

9. (opcional-avançado) Crie um método na classe `Data` que devolva o valor formatado da data, isto é, devolva uma String com "dia/mes/ano". Isso para que o método `recuperaDadosParaImpressao` da classe `Conta` possa ficar assim:

```
class Conta {
    // atributos e metodos

    String recuperaDadosParaImpressao() {
        // imprime outros atributos...
        dados += "\nData de abertura: " + this.dataDeAbertura.formatada();
        return dados;
    }
}
```

4.13 DESAFIOS

1. Um método pode chamar ele mesmo. Chamamos isso de **recursão**. Você pode resolver a série de Fibonacci usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da seguinte maneira:

   ```
   Fibonacci fibonacci = new Fibonacci();
   for (int i = 1; i <= 6; i++) {
       int resultado = fibonacci.calculaFibonacci(i);
       System.out.println(resultado);
   }
   ```

   Aqui imprimirá a sequência de Fibonacci até a sexta posição, isto é: 1, 1, 2, 3, 5, 8.

   Este método `calculaFibonacci` não pode ter nenhum laço, só pode chamar ele mesmo como método. Pense nele como uma função, que usa a própria função para calcular o resultado.

2. Por que o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se usa um laço)?

3. Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o **operador condicional ternário**. (ternary operator)

4.14 FIXANDO O CONHECIMENTO

O objetivo dos exercícios a seguir é fixar o conceito de classes e objetos, métodos e atributos. Dada a estrutura de uma classe, basta traduzi-la para a linguagem Java e fazer uso de um objeto da mesma em um programa simples.

Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos nos pequenos programas abaixo:

- Programa 1

```
Classe: Pessoa
Atributos: nome, idade.
Método: void fazAniversario()
```

Crie uma pessoa, coloque seu nome e idade iniciais, faça alguns aniversários (aumentando a idade) e imprima seu nome e sua idade.

- Programa 2

```
Classe: Porta
Atributos: aberta, cor, dimensaoX, dimensaoY, dimensaoZ
Métodos: void abre()
         void fecha()
         void pinta(String s)
         boolean estaAberta()
```

Crie uma porta, abra e feche a mesma, pinte-a de diversas cores, altere suas dimensões e use o método `estaAberta` para verificar se ela está aberta.

- Programa 3

```
Classe: Casa
Atributos: cor, porta1, porta2, porta3
Método: void pinta(String s),
        int quantasPortasEstaoAbertas()
```

Crie uma casa e pinte-a. Crie três portas e coloque-as na casa; abra e feche as mesmas como desejar. Utilize o método `quantasPortasEstaoAbertas` para imprimir o número de portas abertas.

CAPÍTULO 5

MODIFICADORES DE ACESSO E ATRIBUTOS DE CLASSE

"A marca do homem imaturo é que ele quer morrer nobremente por uma causa, enquanto a marca do homem maduro é querer viver modestamente por uma."--J. D. Salinger

Ao término desse capítulo, você será capaz de:

- controlar o acesso aos seus métodos, atributos e construtores através dos modificadores private e public;
- escrever métodos de acesso a atributos do tipo getters e setters;
- escrever construtores para suas classes;
- utilizar variáveis e métodos estáticos.

5.1 CONTROLANDO O ACESSO

Um dos problemas mais simples que temos no nosso sistema de contas é que o método `saca` permite sacar mesmo que o saldo seja insuficiente. A seguir você pode lembrar como está a classe `Conta`:

```
class Conta {
    String titular;
    int numero;
    double saldo;

    // ..

    void saca(double valor) {
        this.saldo = this.saldo - valor;
    }
}
```

A classe a seguir mostra como é possível ultrapassar o limite de saque usando o método `saca`:

```
class TestaContaEstouro1 {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 1000.0;
        minhaConta.saca(50000); // saldo é só 1000!!
    }
}
```

Podemos incluir um `if` dentro do nosso método `saca()` para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo abaixo de 0. Fizemos isso no capítulo de orientação a objetos básica.

Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta. O código a seguir faz isso diretamente:

```
class TestaContaEstouro2 {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = -200; //saldo está abaixo de 0
    }
}
```

Como evitar isso? Uma ideia simples seria testar se não estamos sacando um valor maior que o saldo toda vez que formos alterá-lo:

```
class TestaContaEstouro3 {

    public static void main(String[] args) {
        // a Conta
        Conta minhaConta = new Conta();
        minhaConta.saldo = 100;

        // quero mudar o saldo para -200
        double novoSaldo = -200;

        // testa se o novoSaldo é válido
        if (novoSaldo < 0) { //
            System.out.println("Não posso mudar para esse saldo");
        } else {
            minhaConta.saldo = novoSaldo;
        }
    }
}
```

Esse código iria se repetir ao longo de toda nossa aplicação e, pior, alguém pode esquecer de fazer essa comparação em algum momento, deixando a conta na situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe `Conta` a invocar o método `saca` e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.

Para fazer isso no Java, basta declarar que os atributos não podem ser acessados de fora da classe através da palavra chave `private` :

```
class Conta {
    private double saldo;
    // ...
}
```

`private` é um **modificador de acesso** (também chamado de **modificador de visibilidade**).

Marcando um atributo como privado, fechamos o acesso ao mesmo em relação a todas as outras

classes, fazendo com que o seguinte código não compile:

```
class TestaAcessoDireto {
   public static void main(String[] args) {
    Conta minhaConta = new Conta();
    //não compila! você não pode acessar o atributo privado de outra classe
    minhaConta.saldo = 1000;
   }
}
```

```
    TesteAcessoDireto.java:5 saldo has private access in Conta
                                minhaConta.saldo = 1000;
                                           ^
    1 error
```

Na orientação a objetos, é prática quase que obrigatória proteger seus atributos com `private`. (discutiremos outros modificadores de acesso em outros capítulos).

Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não! Esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema. Muitas outras vezes nem mesmo queremos que outras classes saibam da existência de determinado atributo, escondendo-o por completo, já que ele diz respeito ao funcionamento interno do objeto.

Repare que, quem invoca o método `saca` não faz a menor ideia de que existe uma verificação para o valor do saque. Para quem for usar essa classe, basta saber o que o método faz e não como exatamente ele o faz (o que um método faz é sempre mais importante do que como ele faz: mudar a implementação é fácil, já mudar a *assinatura* de um método vai gerar problemas).

A palavra chave `private` também pode ser usada para modificar o acesso a um método. Tal funcionalidade é utilizada em diversos cenários: quando existe um método que serve apenas para auxiliar a própria classe e quando há código repetido dentro de dois métodos da classe são os mais comuns. Sempre devemos expôr o mínimo possível de funcionalidades, para criar um baixo acoplamento entre as nossas classes.

Da mesma maneira que temos o `private`, temos o modificador `public`, que permite a todos acessarem um determinado atributo ou método :

```
class Conta {
    //...
    public void saca(double valor) {
        //posso sacar até saldo
        if (valor > this.saldo){
            System.out.println("Não posso sacar um valor maior que o saldo!");
        } else {
            this.saldo = this.saldo - valor;
        }
    }
}
```

E QUANDO NÃO HÁ MODIFICADOR DE ACESSO?

Até agora, tínhamos declarado variáveis e métodos sem nenhum modificador como `private` e `public`. Quando isto acontece, o seu método ou atributo fica num estado de visibilidade intermediário entre o `private` e o `public`, que veremos mais pra frente, no capítulo de pacotes.

É muito comum, e faz todo sentido, que seus atributos sejam `private` e quase todos seus métodos sejam `public` (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu!

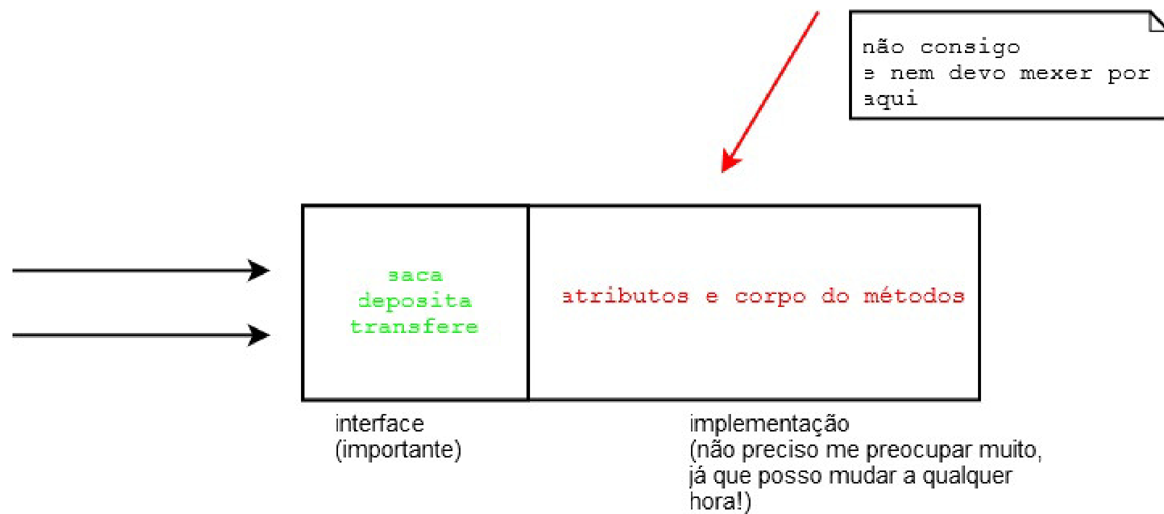
Melhor ainda! O dia em que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca`, o que faz pleno sentido. Como exemplo, imagine cobrar CPMF de cada saque: basta você modificar ali, e nenhum outro código, fora a classe `Conta`, precisará ser recompilado. Mais: as classes que usam esse método nem precisam ficar sabendo de tal modificação! Você precisa apenas recompilar aquela classe e substituir aquele arquivo `.class`. Ganhamos muito em esconder o funcionamento do nosso método na hora de dar manutenção e fazer modificações.

5.2 ENCAPSULAMENTO

O que começamos a ver nesse capítulo é a ideia de **encapsular**, isto é, esconder todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

Encapsular é **fundamental** para que seu sistema seja suscetível a mudanças: não precisaremos

mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está **encapsulada**. (veja o caso do método saca)



O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

Programando voltado para a interface e não para a implementação

É sempre bom programar pensando na interface da sua classe, como seus usuários a estarão utilizando, e não somente em como ela vai funcionar.

A implementação em si, o conteúdo dos métodos, não tem tanta importância para o usuário dessa classe,uma vez que ele só precisa saber o que cada método pretende fazer, e não como ele faz, pois isto pode mudar com o tempo.

Essa frase vem do livro Design Patterns, de Eric Gamma et al. Um livro cultuado no meio da orientação a objetos.

Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:

- Quando você dirige um carro, o que te importa são os pedais e o volante (interface) e não o motor que você está usando (implementação). É claro que um motor diferente pode te dar melhores resultados, mas **o que ele faz** é o mesmo que um motor menos potente, a diferença está em **como ele faz**. Para trocar um carro a álcool para um a gasolina você não precisa reaprender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem usando eles da mesma maneira).

- Todos os celulares fazem a mesma coisa (interface), eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem (implementação), mas repare que para efetuar uma ligação pouco importa se o celular é iPhone ou Android, isso fica encapsulado na implementação (que aqui são os circuitos).

Já temos conhecimentos suficientes para resolver aquele problema da validação de CPF:

```
class Cliente {
    private String nome;
    private String endereco;
    private String cpf;
    private int idade;

    public void mudaCPF(String cpf) {
        validaCPF(cpf);
        this.cpf = cpf;
    }

    private void validaCPF(String cpf) {
        // série de regras aqui, falha caso não seja válido
    }

    // ..
}
```

Se alguém tentar criar um `Cliente` e não usar o `mudaCPF` para alterar um `cpf` diretamente, vai receber um erro de compilação, já que o atributo `CPF` é **privado**. E o dia que você não precisar verificar o CPF de quem tem mais de 60 anos? Seu método fica o seguinte:

```
public void mudaCPF(String cpf) {
    if (this.idade <= 60) {
        validaCPF(cpf);
    }
    this.cpf = cpf;
}
```

O controle sobre o `CPF` está centralizado: ninguém consegue acessá-lo sem passar por aí, a classe `Cliente` é a única responsável pelos seus próprios atributos!

5.3 GETTERS E SETTERS

O modificador `private` faz com que ninguém consiga modificar, nem mesmo ler, o atributo em questão. Com isso, temos um problema: como fazer para mostrar o `saldo` de uma `Conta`, já que nem mesmo podemos acessá-lo para leitura?

Precisamos então arranjar **uma maneira de** fazer esse acesso. Sempre que precisamos arrumar **uma maneira de fazer alguma coisa com um objeto**, utilizamos de métodos! Vamos então criar um método, digamos `pegaSaldo`, para realizar essa simples tarefa:

```
class Conta {
```

```
    private double saldo;

    // outros atributos omitidos

    public double pegaSaldo() {
        return this.saldo;
    }

    // deposita() e saca() omitidos
}
```

Para acessarmos o saldo de uma conta, podemos fazer:

```
class TestaAcessoComPegaSaldo {
   public static void main(String[] args) {
    Conta minhaConta = new Conta();
    minhaConta.deposita(1000);
    System.out.println("Saldo: " + minhaConta.pegaSaldo());
   }
}
```

Para permitir o acesso aos atributos (já que eles são `private`) de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor.

A convenção para esses métodos é de colocar a palavra `get` ou `set` antes do nome do atributo. Por exemplo, a nossa conta com `saldo` , `limite` e `titular` fica assim, no caso da gente desejar dar acesso a leitura e escrita a todos os atributos:

```
class Conta {

    private String titular;
    private double saldo;

    public double getSaldo() {
        return this.saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }

    public String getTitular() {
        return this.titular;
    }

    public void setTitular(String titular) {
        this.titular = titular;
    }
}
```

É uma má prática criar uma classe e, logo em seguida, criar getters e setters para todos seus atributos. Você só deve criar um getter ou setter se tiver a real necessidade. Repare que nesse exemplo `setSaldo` não deveria ter sido criado, já que queremos que todos usem `deposita()` e `saca()` .

Outro detalhe importante, um método `getX` não necessariamente retorna o valor de um atributo que chama `X` do objeto em questão. Isso é interessante para o encapsulamento. Imagine a situação:

queremos que o banco sempre mostre como `saldo` o valor do limite somado ao saldo (uma prática comum dos bancos que costuma iludir seus clientes). Poderíamos sempre chamar `c.getLimite() + c.getSaldo()`, mas isso poderia gerar uma situação de "replace all" quando precisássemos mudar como o saldo é mostrado. Podemos encapsular isso em um método e, porque não, dentro do próprio `getSaldo`? Repare:

```
class Conta {

    private String titular;
    private double saldo;
    private double limite; // adicionando um limite a conta

    public double getSaldo() {
        return this.saldo + this.limite;
    }

    // deposita() saca() e transfere() omitidos

    public String getTitular() {
        return this.titular;
    }

    public void setTitular(String titular) {
        this.titular = titular;
    }
}
```

O código acima nem possibilita a chamada do método `getLimite()`, ele não existe. E nem deve existir enquanto não houver essa necessidade. O método `getSaldo()` não devolve simplesmente o `saldo`... e sim o que queremos que seja mostrado como se fosse o `saldo`. Utilizar getters e setters não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de encapsulamento, pois esconde a maneira como os objetos guardam seus dados. É uma prática muito importante.

Nossa classe está totalmente pronta? Isto é, existe a chance dela ficar com saldo menor que 0? Pode parecer que não, mas, e se depositarmos um valor negativo na conta? Ficaríamos com menos dinheiro que o permitido, já que não esperávamos por isso. Para nos proteger disso basta mudarmos o método `deposita()` para que ele verifique se o valor é necessariamente positivo.

Depois disso precisaríamos mudar mais algum outro código? A resposta é não, graças ao encapsulamento dos nossos dados.

CUIDADO COM OS GETTERS E SETTERS!

Como já dito, não devemos criar getters e setters sem um motivo explicito. No blog da Caelum há um artigo que ilustra bem esses casos:

http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/

5.4 CONSTRUTORES

Quando usamos a palavra chave `new` , estamos construindo um objeto. Sempre quando o `new` é chamado, ele executa o **construtor da classe**. O construtor da classe é um bloco declarado com o **mesmo nome** que a classe:

```
class Conta {
    String titular;
    int numero;
    double saldo;

    // construtor
    Conta() {
        System.out.println("Construindo uma conta.");
    }

    // ..
}
```

Então, quando fizermos:

```
Conta c = new Conta();
```

A mensagem "construindo uma conta" aparecerá. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. Um construtor pode parecer, mas **não é** um método.

O CONSTRUTOR DEFAULT

Até agora, as nossas classes não possuíam nenhum construtor. Então como é que era possível dar `new` , se todo `new` chama um construtor **obrigatoriamente**?

Quando você não declara nenhum construtor na sua classe, o Java cria um para você. Esse construtor é o **construtor default**, ele não recebe nenhum argumento e o corpo dele é vazio.

A partir do momento que você declara um construtor, o construtor default não é mais fornecido.

O interessante é que um construtor pode receber um argumento, podendo assim inicializar algum

tipo de informação:

```
class Conta {
    String titular;
    int numero;
    double saldo;

    // construtor
    Conta(String titular) {
        this.titular = titular;
    }

    // ..
}
```

Esse construtor recebe o titular da conta. Assim, quando criarmos uma conta, ela já terá um determinado titular.

```
String carlos = "Carlos";
Conta c = new Conta(carlos);
System.out.println(c.titular);
```


5.5 A NECESSIDADE DE UM CONSTRUTOR

Tudo estava funcionando até agora. Para que utilizamos um construtor?

A ideia é bem simples. Se toda conta precisa de um titular, como obrigar todos os objetos que forem criados a ter um valor desse tipo? Basta criar um único construtor que recebe essa String!

O construtor se resume a isso! Dar possibilidades ou obrigar o usuário de uma classe a passar argumentos para o objeto durante o processo de criação do mesmo.

Por exemplo, não podemos abrir um arquivo para leitura sem dizer qual é o nome do arquivo que desejamos ler! Portanto, nada mais natural que passar uma `String` representando o nome de um arquivo na hora de criar um objeto do tipo de leitura de arquivo, e que isso seja obrigatório.

Você pode ter mais de um construtor na sua classe e, no momento do `new`, o construtor apropriado será escolhido.

Construtor: um método especial?

Um construtor **não é** um método. Algumas pessoas o chamam de um método especial, mas definitivamente não é, já que não possui retorno e só é chamado durante a construção do objeto.

Chamando outro construtor

Um construtor só pode rodar durante a construção do objeto, isto é, você nunca conseguirá chamar o construtor em um objeto já construído. Porém, durante a construção de um objeto, você pode fazer com que um construtor chame outro, para não ter de ficar copiando e colando:

```
class Conta {
    String titular;
    int numero;
    double saldo;

    // construtor
    Conta (String titular) {
        //     faz mais uma série de inicializações e configurações
        this.titular = titular;
    }

    Conta (int numero, String titular) {
        this(titular); // chama o construtor que foi declarado acima
        this.numero = numero;
    }

    //..
}
```

Existe um outro motivo, o outro lado dos construtores: facilidade. Às vezes, criamos um construtor que recebe diversos argumentos para não obrigar o usuário de uma classe a chamar diversos métodos do tipo `'set'`.

No nosso exemplo do CPF, podemos forçar que a classe `Cliente` receba no mínimo o CPF, dessa maneira um `Cliente` já será construído e com um CPF válido.

JAVA BEAN

Quando criamos uma classe com todos os atributos privados, seus getters e setters e um construtor vazio (padrão), na verdade estamos criando um Java Bean (mas não confunda com EJB, que é Enterprise Java Beans).

5.6 ATRIBUTOS DE CLASSE

Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isto? A ideia mais simples:

```
Conta c = new Conta();
totalDeContas = totalDeContas + 1;
```

Aqui, voltamos em um problema parecido com o da validação de CPF. Estamos espalhando um código por toda aplicação, e quem garante que vamos conseguir lembrar de incrementar a variável `totalDeContas` toda vez?

Tentamos então, passar para a seguinte proposta:

```
class Conta {
    private int totalDeContas;
    //...

    Conta() {
        this.totalDeContas = this.totalDeContas + 1;
    }
}
```

Quando criarmos duas contas, qual será o valor do `totalDeContas` de cada uma delas? Vai ser 1. Pois cada uma tem essa variável. **O atributo é de cada objeto**.

Seria interessante então, que essa variável fosse **única**, compartilhada por todos os objetos dessa classe. Dessa maneira, quando mudasse através de um objeto, o outro enxergaria o mesmo valor. Para fazer isso em java, declaramos a variável como `static` .

```
private static int totalDeContas;
```

Quando declaramos um atributo como `static` , ele passa a não ser mais um atributo de cada objeto, e sim um **atributo da classe**, a informação fica guardada pela classe, não é mais individual para cada objeto.

Para acessarmos um atributo estático, não usamos a palavra chave `this` , mas sim o nome da classe:

```
class Conta {
    private static int totalDeContas;
    //...
```

```
    Conta() {
        Conta.totalDeContas = Conta.totalDeContas + 1;
    }
}
```

Já que o atributo é privado, como podemos acessar essa informação a partir de outra classe? Precisamos de um getter para ele!

```
class Conta {
    private static int totalDeContas;
    //...

    Conta() {
        Conta.totalDeContas = Conta.totalDeContas + 1;
    }

    public int getTotalDeContas() {
        return Conta.totalDeContas;
    }
}
```

Como fazemos então para saber quantas contas foram criadas?

```
Conta c = new Conta();
int total = c.getTotalDeContas();
```

Precisamos criar uma conta antes de chamar o método! Isso não é legal, pois gostaríamos de saber quantas contas existem sem precisar ter acesso a um objeto conta. A ideia aqui é a mesma, transformar esse método que todo objeto conta tem em um método de toda a classe. Usamos a palavra `static` de novo, mudando o método anterior.

```
public static int getTotalDeContas() {
    return Conta.totalDeContas;
}
```

Para acessar esse novo método:

```
int total = Conta.getTotalDeContas();
```

Repare que estamos chamando um método não com uma referência para uma `Conta` , e sim usando o nome da classe.

MÉTODOS E ATRIBUTOS ESTÁTICOS

Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe, o que faz todo sentido já que dentro de um método estático não temos acesso à referência `this` , pois um método estático é chamado através da classe, e não de um objeto.

O `static` realmente traz um "cheiro" procedural, porém em muitas vezes é necessário.

5.7 UM POUCO MAIS...

- Em algumas empresas, o UML é amplamente utilizado. Às vezes, o programador recebe o UML já pronto, completo, e só deve preencher a implementação, devendo seguir à risca o UML. O que você acha dessa prática? Quais as vantagens e desvantagens.
- Se uma classe só tem atributos e métodos estáticos, que conclusões podemos tirar? O que lhe parece um método estático em casos como esses?
- No caso de atributos booleanos, pode-se usar no lugar do `get` o sufixo `is` . Dessa maneira, caso tivéssemos um atributo booleano `ligado` , em vez de `getLigado` poderíamos ter `isLigado` .

5.8 EXERCÍCIOS: ENCAPSULAMENTO, CONSTRUTORES E STATIC

1. O que é necessário fazer para **garantirmos** que os atributos da classe Conta não sejam acessados de forma direta em outra classe que não seja a própria classe Conta?

2. Após ter deixado os atributos da classe Conta com acesso restrito (privado), tente criar uma `Conta` na classe `TestaConta` , dentro do `main` e modificar ou ler os atributos da conta criada. O que acontece?

   Crie apenas os getters e setters necessários na sua classe `Conta` . Pense sempre se é preciso criar cada um deles.

   Não copie e cole! Aproveite para praticar a sintaxe. Logo passaremos a usar o Eclipse e aí sim teremos procedimentos mais simples para este tipo de tarefa.

   Repare que o método `calculaRendimento` parece também um getter. Aliás, seria comum alguém nomeá-lo de `getRendimento` . Getters não precisam apenas retornar atributos. Eles podem trabalhar

com esses dados.

3. Altere suas classes que acessam e modificam atributos de uma `Conta` para utilizar os getters e setters recém criados.

4. Faça com que sua classe `Conta` possa receber, opcionalmente, o nome do titular da `Conta` durante a criação do objeto.

5. (opcional) Adicione um atributo na classe `Conta` de tipo `int` que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo `Conta` . A primeira `Conta` instanciada tem identificador 1, a segunda 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.

   Crie um getter para o identificador. Devemos ter um setter?

6. (opcional) Como garantir que datas como 31/2/2021 não sejam aceitas pela sua classe `Data` ?

7. (opcional) Suponha que temos a classe `PessoaFisica` que tem um CPF como atributo. Como garantir que pessoa física alguma tenha CPF invalido, nem seja criada `PessoaFisica` sem cpf inicial? (Suponha que já existe um algoritmo de validação de cpf: basta passar o cpf por um método `valida(String x)...`)

5.9 DESAFIOS

1. Porque esse código não compila?

```
class Teste {
    int x = 37;
    public static void main(String [] args) {
        System.out.println(x);
    }
}
```

2. Imagine que tenha uma classe `FabricaDeCarro` e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em Java, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso? (pesquise: singleton design pattern)

Capítulo 6

ECLIPSE IDE

"Dá-se importância aos antepassados quando já não temos nenhum."--François Chateaubriand

Neste capítulo, você será apresentado ao Ambiente de Desenvolvimento Eclipse e suas principais funcionalidades.

6.1 O ECLIPSE

O Eclipse (http://www.eclipse.org) é uma IDE (integrated development environment). Diferente de uma RAD, onde o objetivo é desenvolver o mais rápido possível através do *arrastar-e-soltar do mouse*, onde montanhas de código são gerados em background, uma IDE te auxilia no desenvolvimento, evitando se intrometer e fazer muita mágica.

O Eclipse é a IDE líder de mercado. Formada por um consórcio liderado pela IBM, possui seu código livre.

Veremos aqui os principais recursos do Eclipse. Você perceberá que ele evita ao máximo te atrapalhar e apenas gera trechos de códigos óbvios, sempre ao seu comando. Existem também centenas de plugins gratuitos para gerar diagramas UML, suporte a servidores de aplicação, visualizadores de banco de dados e muitos outros.

Baixe o Eclipse do site oficial http://www.eclipse.org. Apesar de ser escrito em Java, a biblioteca gráfica usada no Eclipse, chamada SWT, usa componentes nativos do sistema operacional. Por isso você deve baixar a versão correspondente ao seu sistema operacional.

Descompacte o arquivo e pronto, basta rodar o executável.

Outras IDEs

Uma outra IDE open source famosa é o Netbeans, da Oracle. (http://www.netbeans.org).

Além dessas, Oracle, Borland e a própria IBM possuem IDEs comerciais e algumas versões mais restritas de uso livre.

A empresa JetBrains desenvolve o IntelliJ IDEA, uma IDE paga que tem ganho muitos adeptos.

Você pode também fazer o curso data dessa apostila na Caelum



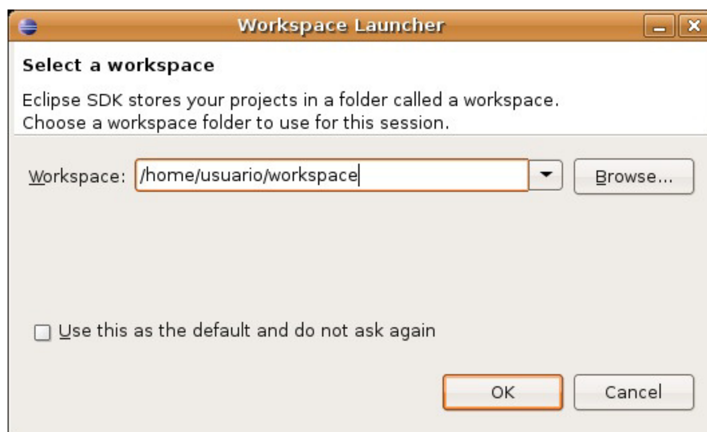
Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

6.2 APRESENTANDO O ECLIPSE

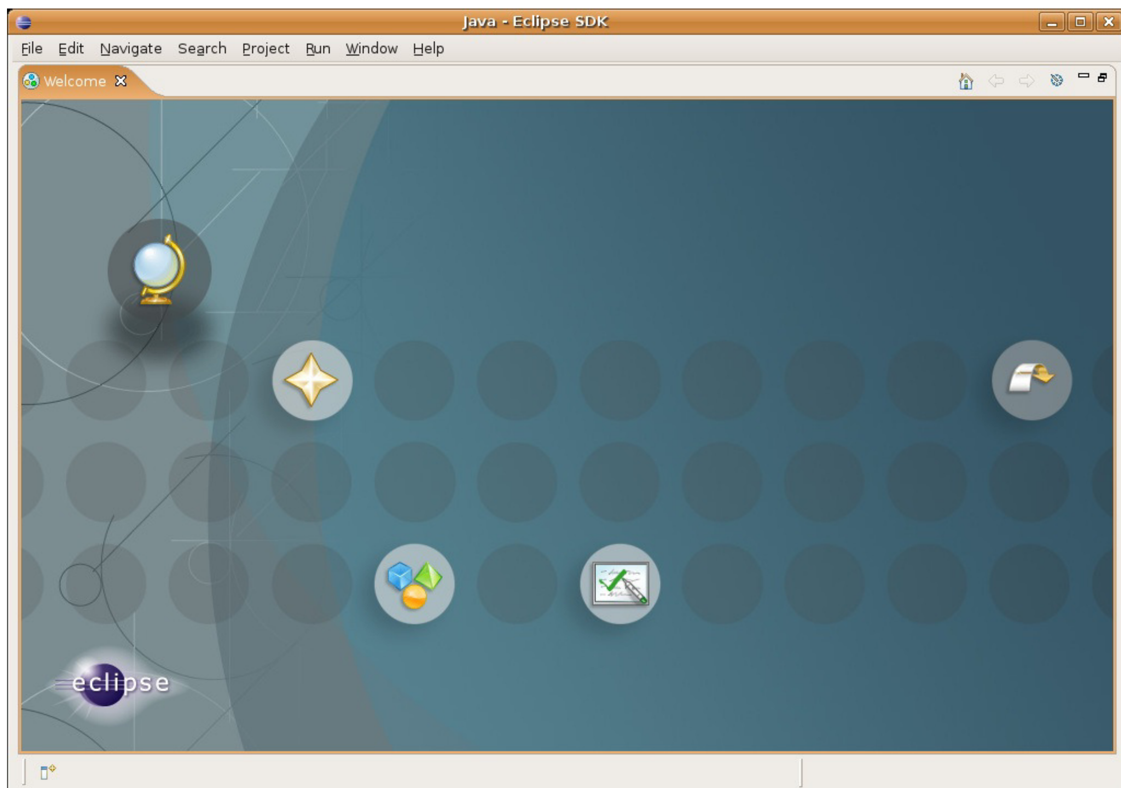
Clique no ícone do Eclipse no seu Desktop.

A primeira pergunta que ele te faz é que workspace você vai usar. Workspace define o diretório em que as suas configurações pessoais e seus projetos serão gravados.



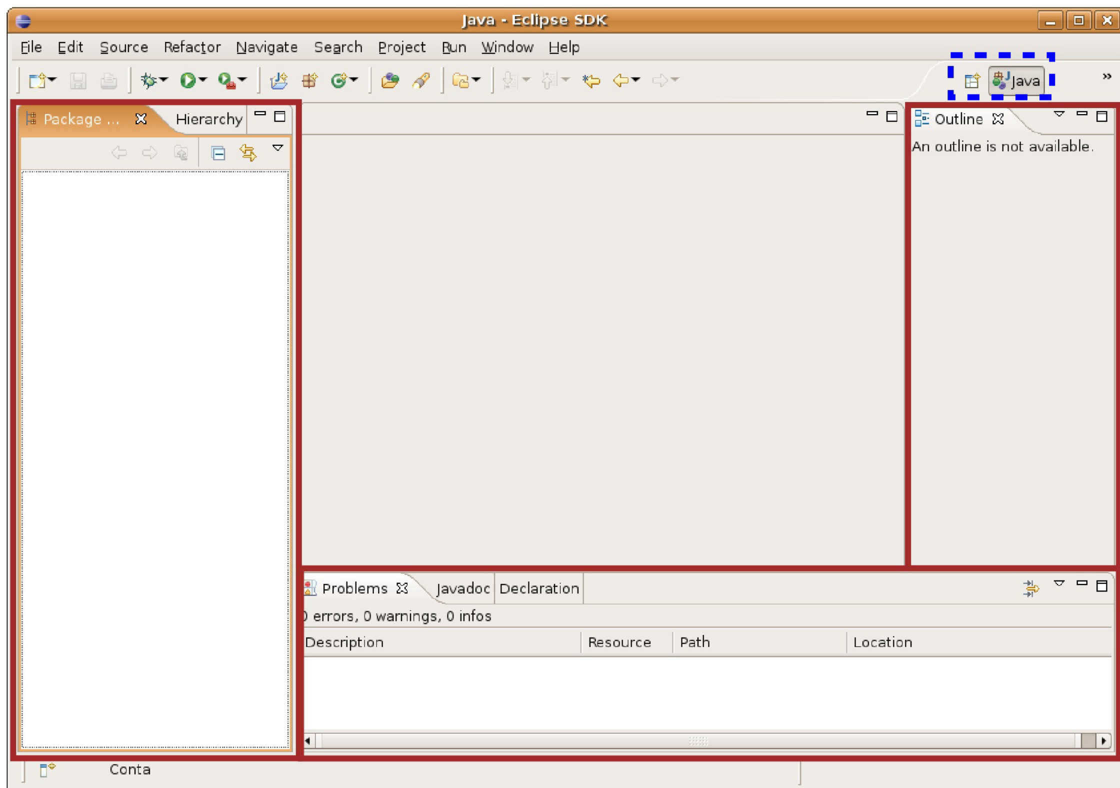
Você pode deixar o diretório pré-definido.

Logo em seguida, uma tela de Welcome será aberta, onde você tem diversos links para tutoriais e ajuda. Clique em Workbench.

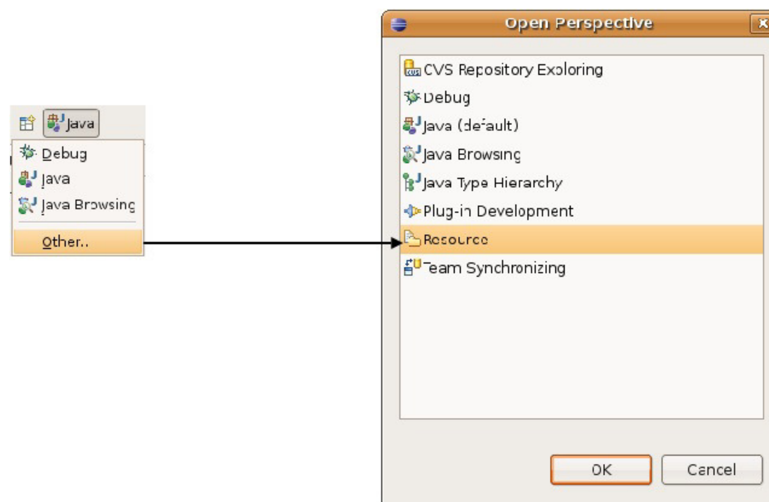


6.3 VIEWS E PERSPECTIVE

Feche a tela de Welcome e você verá a tela abaixo. Nesta tela, destacamos as Views (em linha contínua) e as Perspectives (em linha pontilhada) do Eclipse.



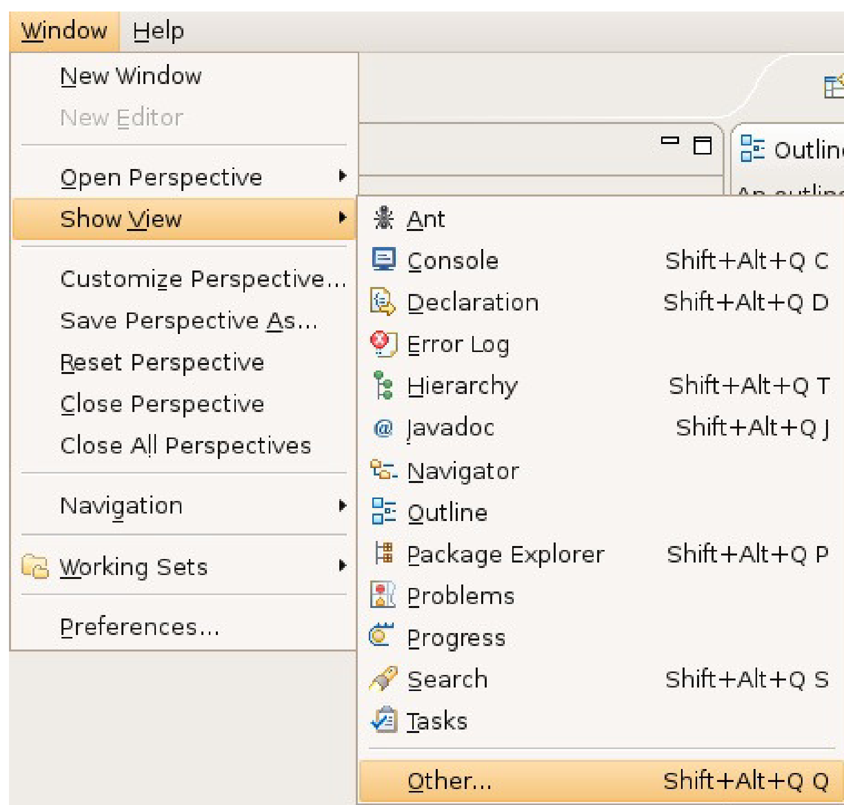
Mude para a perspectiva Resource, clicando no ícone ao lado da perspectiva Java, selecionando Other e depois Resource. Neste momento, trabalharemos com esta perspectiva, antes da de Java, pois ela possui um conjunto de Views mais simples.



A View Navigator mostra a estrutura de diretório assim como está no sistema de arquivos. A View Outline mostra um resumo das classes, interfaces e enumerações declaradas no arquivo java atualmente editado (serve também para outros tipos de arquivos).

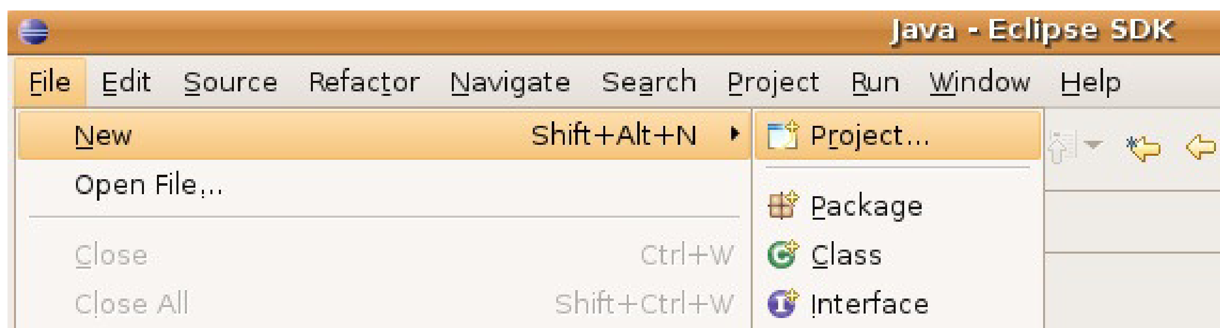
No menu **Window -> Show View -> Other**, você pode ver as dezenas de Views que já vem

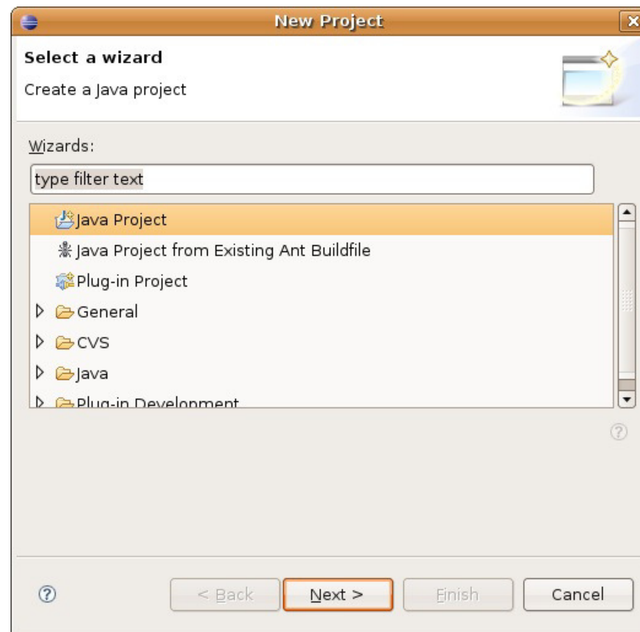
embutidas no Eclipse. Acostume-se a sempre procurar novas Views, elas podem te ajudar em diversas tarefas.



6.4 CRIANDO UM PROJETO NOVO

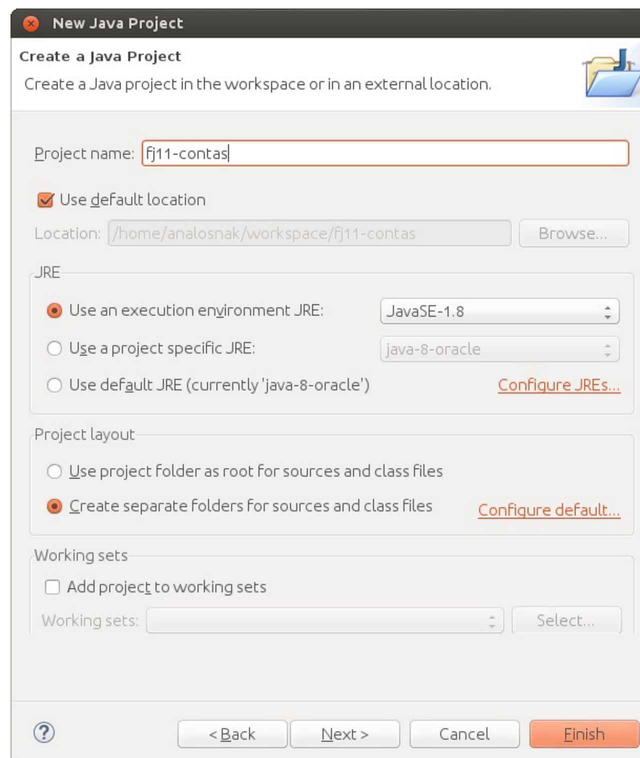
Vá em **File -> New -> Project**. Seleciona Java Project e clique em Next.





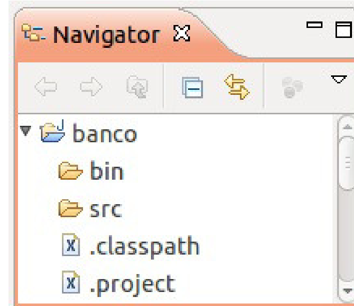
Crie um projeto chamado `fj11-contas` .

Você pode chegar nessa mesma tela clicando com o botão da direta no espaço da View Navigator e seguindo o mesmo menu. Nesta tela, configure seu projeto como na tela abaixo:

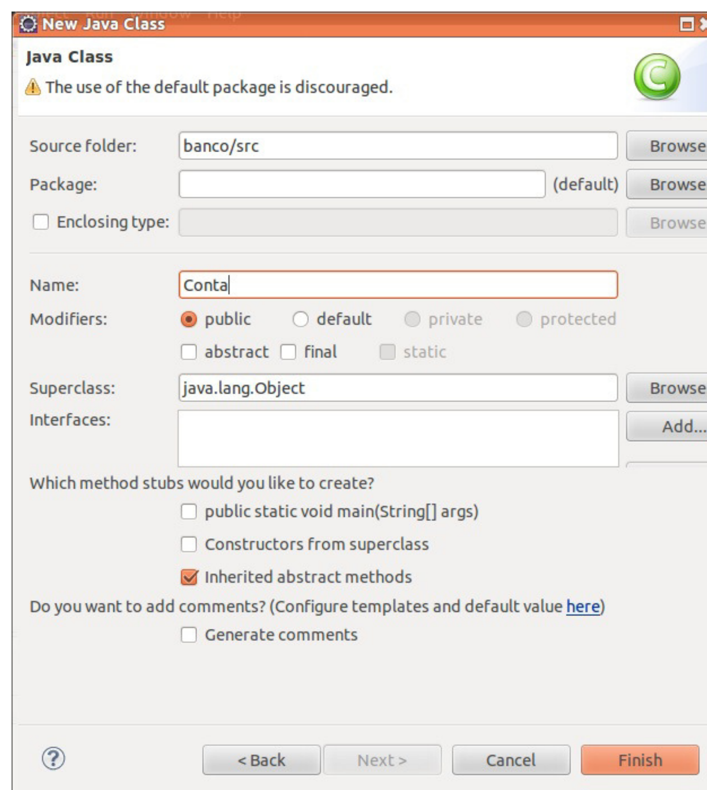


Isto é, marque "create separate source and output folders", desta maneira seus arquivos java e arquivos class estarão em diretórios diferentes, para você trabalhar de uma maneira mais organizada.

Clique em Finish. O Eclipse pedirá para trocar a perspectiva para Java; escolha "**No**" para permanecer em Resource. Na View *Navigator*, você verá o novo projeto e suas pastas e arquivos:

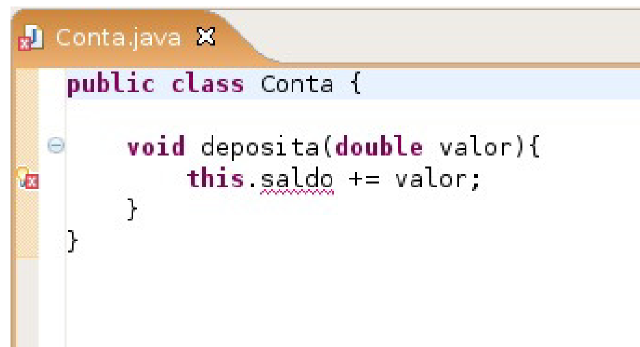


Vamos iniciar nosso projeto criando a classe Conta. Para isso, vá em File -> New -> Other -> Class. Clique em Next e crie a classe seguindo a tela abaixo:

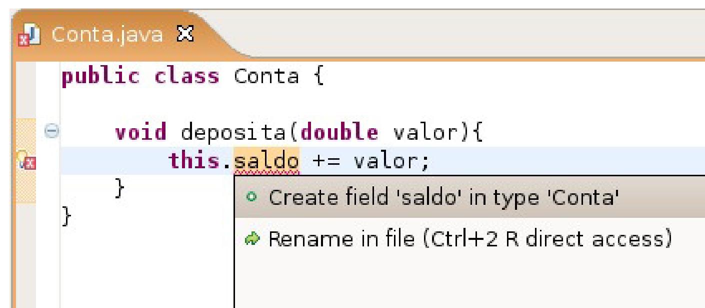


Clique em Finish. O Eclipse possui diversos wizards, mas usaremos o mínimo deles. O interessante é usar o *code assist* e *quick fixes* que a ferramenta possui e veremos em seguida. Não se atente às milhares de opções de cada wizard, a parte mais interessante do Eclipse não é essa.

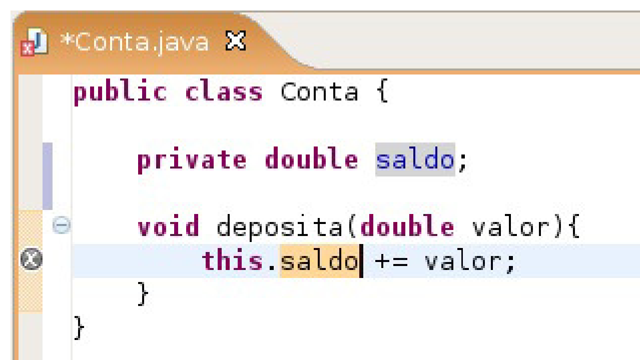
Escreva o método `deposita` como abaixo e note que o Eclipse reclama de erro em `this.saldo` pois este atributo não existe.



Vamos usar o recurso do Eclipse de **quick fix**. Coloque o cursor em cima do erro e aperte Ctrl + 1.

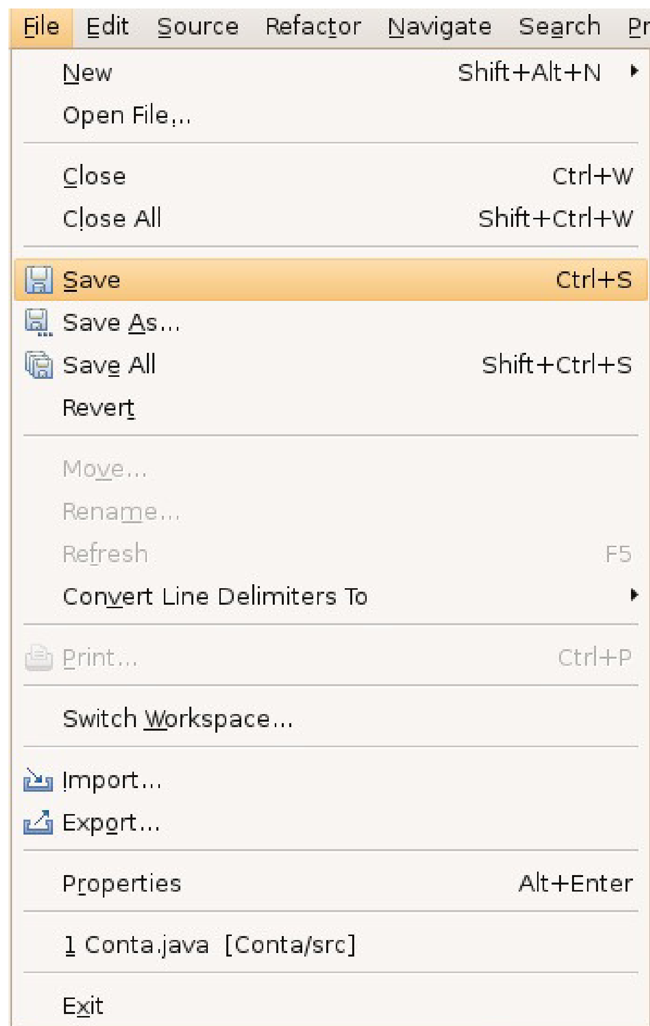


O Eclipse sugerirá possíveis formas de consertar o erro; uma delas é, justamente, criar o campo `saldo` na classe `Conta`, que é nosso objetivo. Clique nesta opção.



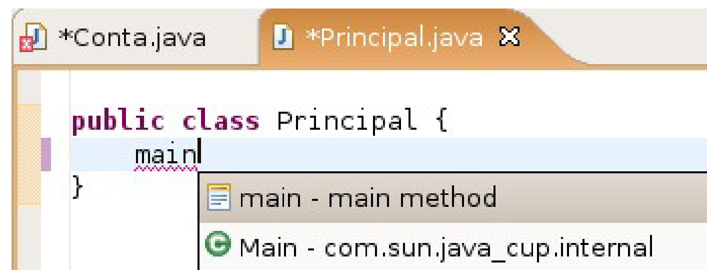
Este recurso de quick fixes, acessível pelo Ctrl+1, é uma das grandes facilidades do Eclipse e é extremamente poderoso. Através dele é possível corrigir boa parte dos erros na hora de programar e, como fizemos, economizar a digitação de certos códigos repetitivos. No nosso exemplo, não precisamos criar o campo antes; o Eclipse faz isso para nós. Ele até acerta a tipagem, já que estamos somando ele a um double. O `private` é colocado por motivos que já estudamos.

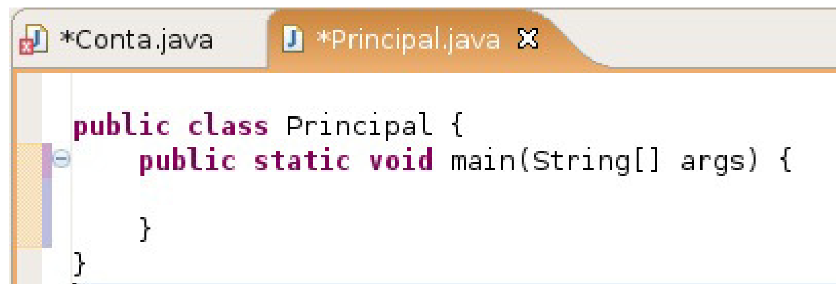
Vá ao menu File -> Save para gravar. Control + S tem o mesmo efeito.



6.5 CRIANDO O MAIN

Crie uma nova classe chamada `Principal` . Vamos colocar um método `main` para testar nossa Conta. Em vez de digitar todo o método `main` , vamos usar o **code assist** do Eclipse. Escreva só `main` e aperte Ctrl + Espaço logo em seguida.





O Eclipse sugerirá a criação do método `main` completo; selecione esta opção. O control + espaço é chamado de **code assist**. Assim como os quick fixes são de extrema importância. Experimente usar o code assist em diversos lugares.

Dentro do método `main` , comece a digitar o seguinte código:

```
Conta conta = new Conta();
conta.deposita(100.0);
```

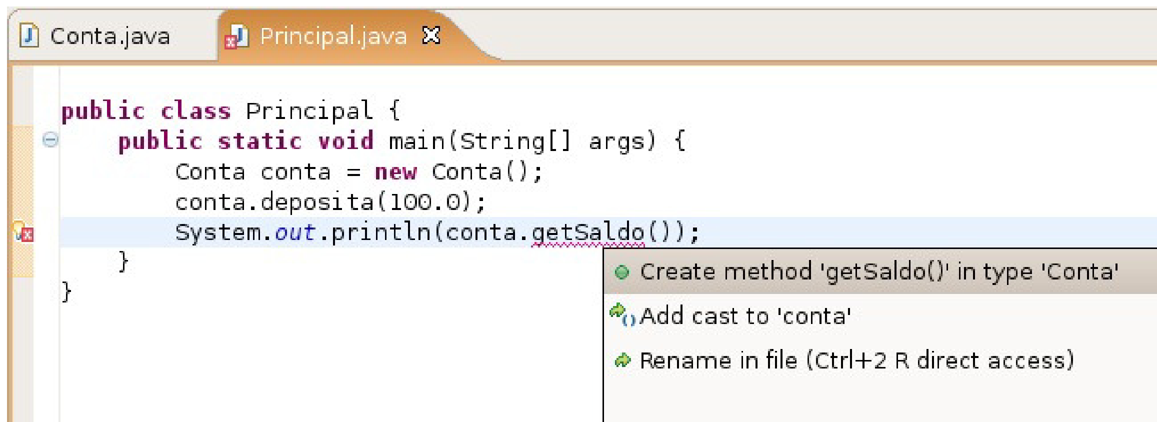
Observe que, na hora de invocar o método sobre o objeto conta, o Eclipse sugere os métodos possíveis. Este recurso é bastante útil, principalmente quando estivermos programando com classes que não são as nossas, como da API do Java. O Eclipse aciona este recurso quando você digita o ponto logo após um objeto (e você pode usar o Ctrl+Espaço para acioná-lo).

Vamos imprimir o saldo com `System.out.println` . Mas, mesmo nesse código, o Eclipse nos ajuda. Escreva `syso` e aperte Ctrl+Espaço que o Eclipse escreverá System.out.println() para você.

Para imprimir, chame o `conta.getSaldo()` :

```
System.out.println(conta.getSaldo());
```

Note que o Eclipse acusará erro em `getSaldo()` porque este método não existe na classe `Conta` . Vamos usar Ctrl+1 em cima do erro para corrigir o problema:



O Eclipse sugere criar um método `getSaldo()` na classe `Conta`. Selecione esta opção e o método será inserido automaticamente.

```
public Object getSaldo() {
    // TODO Auto-generated method stub
    return null;
}
```

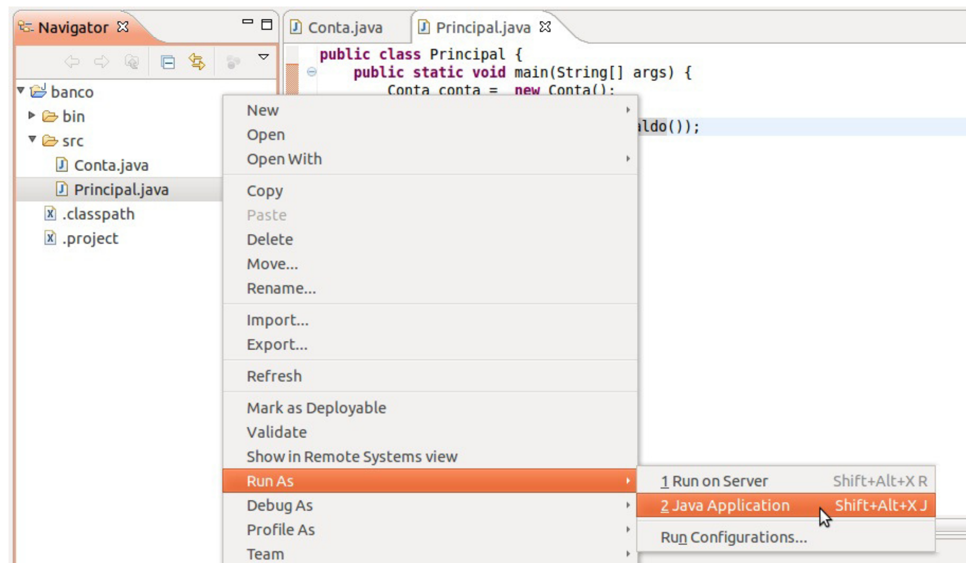
Ele gera um método não exatamente como queríamos, pois nem sempre há como o Eclipse ter de antemão informações suficientes para que ele acerte a assinatura do seu método. Modifique o método `getSaldo` como segue:

```
public double getSaldo() {
    return this.saldo;
}
```

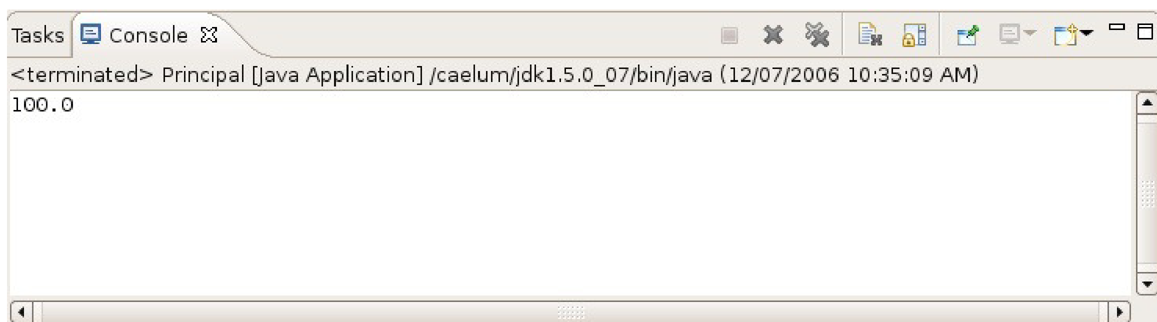
Esses pequenos recursos do Eclipse são de extrema utilidade. Dessa maneira, você pode programar sem se preocupar com métodos que ainda não existem, já que a qualquer momento ele pode gerar o esqueleto (a parte da assinatura do método).

6.6 EXECUTANDO O MAIN

Vamos rodar o método `main` dessa nossa classe. No Eclipse, clique com o botão direito no arquivo `Principal.java` e vá em Run as... Java Application.



O Eclipse abrirá uma View chamada Console onde será apresentada a saída do seu programa:



Quando você precisar rodar de novo, basta clicar no ícone verde de play na toolbar, que roda o programa anterior. Ao lado desse ícone tem uma setinha onde são listados os 10 últimos executados.

6.7 PEQUENOS TRUQUES

O Eclipse possui muitos atalhos úteis para o programador. Sem dúvida os 3 mais importantes de conhecer e de praticar são:

- **Ctrl + 1** Aciona o quick fixes com sugestões para correção de erros.
- **Ctrl + Espaço** Completa códigos
- **Ctrl + 3** Aciona modo de descoberta de menu. Experimente digitar **Ctrl+3** e depois digitar **ggas** e **enter**. Ou então de **Ctrl + 3** e digite *new class*.

Você pode ler muito mais detalhes sobre esses atalhos no blog da Caelum: http://blog.caelum.com.br/as-tres-principais-teclas-de-atalho-do-eclipse/

Existem dezenas de outros. Dentre os mais utilizados pelos desenvolvedores da Caelum, escolhemos os seguintes para comentar:

- **Ctrl + F11** roda a última classe que você rodou. É o mesmo que clicar no ícone verde que parece um botão de play na barra de ferramentas.
- **Ctrl + PgUp** e **Ctrl + PgDown** Navega nas abas abertas. Útil quando estiver editando vários arquivos ao mesmo tempo.
- **Ctrl + Shift + F** Formata o código segundo as convenções do Java
- **Ctrl + M** Expande a View atual para a tela toda (mesmo efeito de dar dois cliques no título da View)
- **Ctrl + Shift + L** Exibe todos os atalhos possíveis.
- **Ctrl + O** Exibe um outline para rápida navegação
- **Alt + Shift + X e depois J** Roda o `main` da classe atual. Péssimo para pressionar! Mais fácil você digitar **Control+3** e depois digitar *Run*!. Abuse desde já do **Control+3**

Veremos mais no decorrer do curso, em especial quando virmos pacotes.

6.8 EXERCÍCIOS: ECLIPSE

1. Crie o projeto `fj11-contas`. Você pode usar o atalho *control + n* ou então ir no menu *File -> New -> Project... -> Java Project*.
2. Dentro do projeto `fj11-contas`, crie a classe `Conta`. Uma conta deve ter as seguintes informações: `saldo` (double), `titular` (String), `numero` (int) e `agencia` (String). Na classe `Conta`, crie os métodos `deposita` e `saca` como nos capítulos anteriores. Crie também uma

classe `TesteDaConta` com o `main` e instancie uma conta. Desta vez, tente abusar do *control + espaço* e *control + 1*.

Por exemplo:

```
publ<ctrl espaco> v<ctrl espaco> deposita(do<ctrl espaço> valor){
```

Repare que até mesmo nomes de variáveis, ele cria para você! Acompanhe as dicas do instrutor.

Muitas vezes, ao criarmos um objeto, nem mesmo declaramos a variável:

```
new Conta();
```

Vá nessa linha e dê *control + 1*. Ele vai sugerir e declarará a variável pra você.

3. Imagine que queremos criar um setter do titular para a classe `Conta`. Dentro da classe `Conta`, digite:

```
setTit<ctrl + espaco>
```

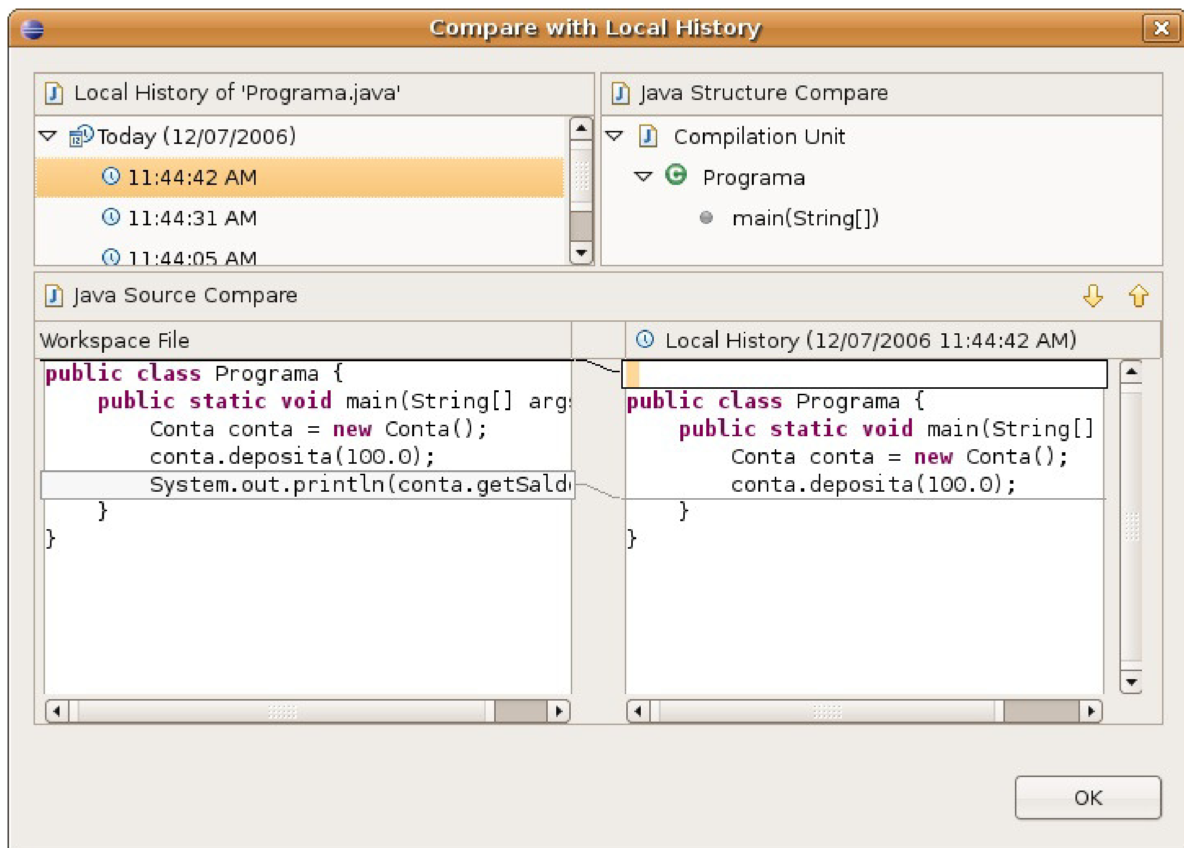
Outra forma para criar os getters e os setters para os atributos da classe `Conta`, é utilizar o atalho *control + 3* e na caixa de seleção digite *ggas*, iniciais de *Generate Getters and Setters*!

OBS: Não crie um setter para o atributo `saldo`!

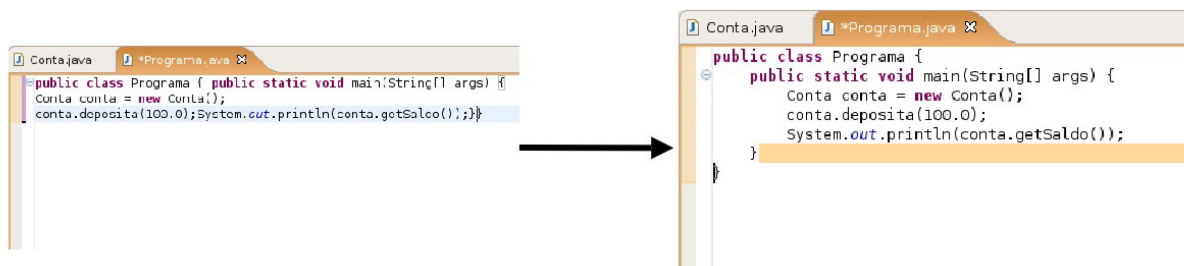
4. Vá para a classe que tem o `main` e segure o CONTROL apertado enquanto você passa o mouse sobre o seu código. Repare que tudo virou hyperlink. Clique em um método que você está invocando na classe `Conta`.

Você pode conseguir o mesmo efeito, de abrir o arquivo no qual o método foi declarado, de uma maneira ainda mais prática: sem usar o mouse, quando o cursor estiver sobre o que você quer analisar, simplesmente clique `F3`.

5. Dê um clique da direita em um arquivo no navigator. Escolha **Compare With -> Local History**. O que é esta tela?

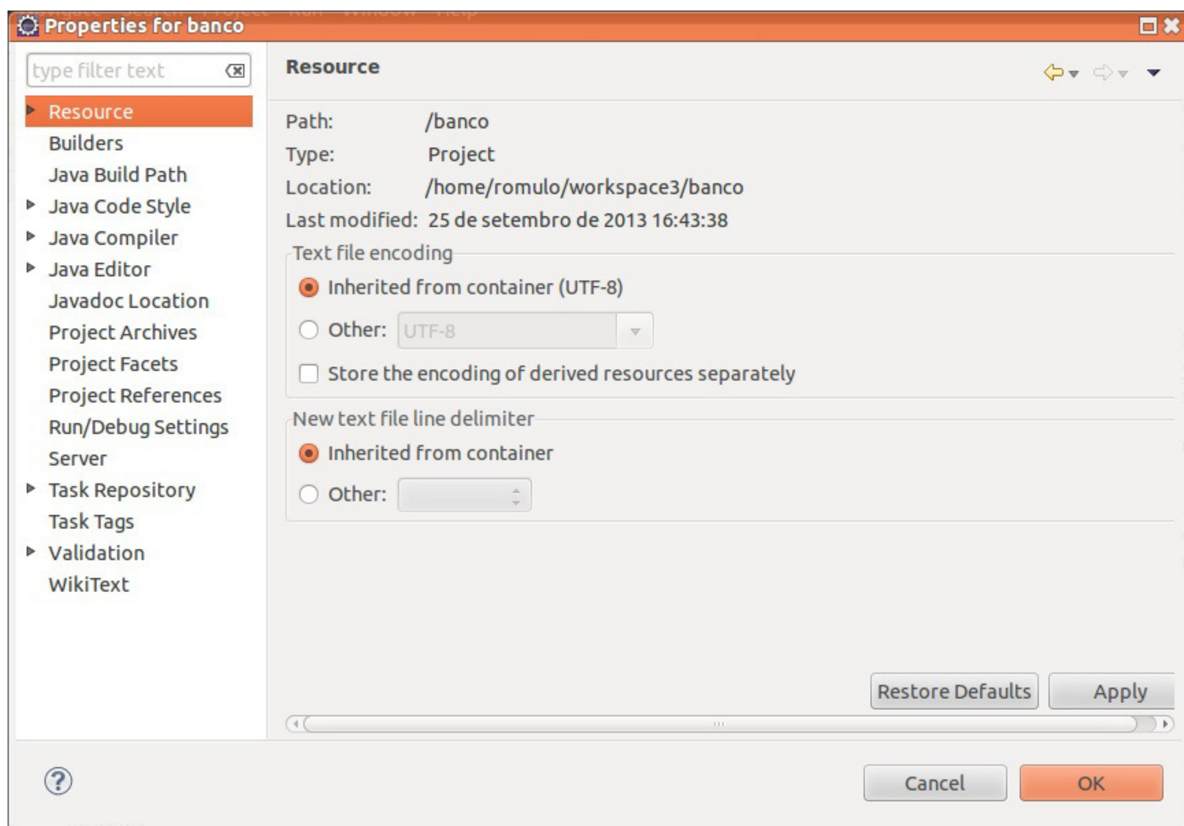


6. Use o *Control* + *Shift* + *F* para formatar o seu código. Dessa maneira, ele vai arrumar a bagunça de espaçamento e enters do seu código.



7. (opcional) O que são os arquivos .project e .classpath? Leia o conteúdo deles.

8. (opcional) Clique da direita no projeto, propriedades. É uma das telas mais importantes do Eclipse, onde você pode configurar diversas informações para o seu projeto, como compilador, versões, formatador e outros.



6.9 DISCUSSÃO EM AULA: REFACTORING

Existe um menu no Eclipse chamado *Refactor*. Ele tem opções bastante interessantes para auxiliar na alteração de código para melhorar organização ou clareza. Por exemplo, uma de suas funcionalidades é tornar possível mudar o nome de uma variável, método ou mesmo classe de forma que uma alteração (em um lugar só do sistema) atualize todas as outras vezes que usavam o nome antigo.

Usar bons nomes no seu código é um excelente começo para mantê-lo legível e fácil de dar manutenção! Mas o assunto "Refatoração" não para por aí: quebrar métodos grandes em menores, dividir classes grandes em algumas pequenas e mais concisas, melhorar o encapsulamento... todas essas são formas de refatoração. E esse menu do Eclipse nos ajuda a fazer várias delas.

CAPÍTULO 7

PACOTES - ORGANIZANDO SUAS CLASSES E BIBLIOTECAS

"Uma discussão prolongada significa que ambas as partes estão erradas" -- Voltaire

Ao término desse capítulo, você será capaz de:

- separar suas classes em pacotes;
- preparar arquivos simples para distribuição.

7.1 ORGANIZAÇÃO

Neste capítulo, aconselhamos que você passe a usar o Eclipse. Você já tem conhecimento suficiente dos erros de compilação do `javac` e agora pode aprender as facilidades que o Eclipse te traz ao ajudar você no código com os chamados quick fixes e quick assists.

Quando um programador utiliza as classes feitas por outro, surge um problema clássico: como escrever duas classes com o mesmo nome?

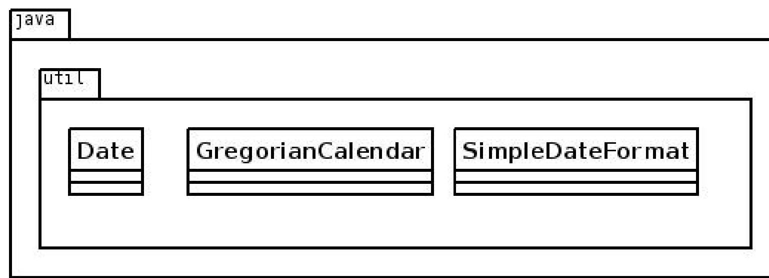
Por exemplo: pode ser que a minha classe de `Data` funcione de um certo jeito, e a classe `Data` de um colega, de outro jeito. Pode ser que a classe de `Data` de uma **biblioteca** funcione ainda de uma terceira maneira diferente.

Como permitir que tudo isso realmente funcione? Como controlar quem quer usar qual classe de `Data` ?

Pensando um pouco mais, notamos a existência de um outro problema e da própria solução: o sistema operacional não permite a existência de dois arquivos com o mesmo nome sob o mesmo diretório, portanto precisamos organizar nossas classes em diretórios diferentes.

Os diretórios estão diretamente relacionados aos chamados **pacotes** e costumam agrupar classes de funcionalidades similares ou relacionadas.

Por exemplo, no pacote `java.util` temos as classes `Date` , `SimpleDateFormat` e `GregorianCalendar` ; todas elas trabalham com datas de formas diferentes.



7.2 DIRETÓRIOS

Se a classe `Cliente` está no pacote `contas`, ela deverá estar no diretório com o mesmo nome: `contas`. Se ela se localiza no pacote `br.com.caelum.contas`, significa que está no diretório **`br/com/caelum/contas`**.

A classe `Cliente`, que se localiza nesse último diretório mencionado, deve ser escrita da seguinte forma:

```
package br.com.caelum.contas;

class Cliente {
    // ...
}
```

Fica fácil notar que a palavra chave `package` indica qual o pacote/diretório contém esta classe.

Um pacote pode conter nenhum ou mais subpacotes e/ou classes dentro dele.

PADRÃO DA NOMENCLATURA DOS PACOTES

O padrão da sun para dar nome aos pacotes é relativo ao nome da empresa que desenvolveu a classe:

```
br.com.nomedaempresa.nomedoprojeto.subpacote
br.com.nomedaempresa.nomedoprojeto.subpacote2
br.com.nomedaempresa.nomedoprojeto.subpacote2.subpacote3
```

Os pacotes só possuem letras minúsculas, não importa quantas palavras estejam contidas nele. Esse padrão existe para evitar ao máximo o conflito de pacotes de empresas diferentes.

As classes do pacote padrão de bibliotecas não seguem essa nomenclatura, que foi dada para bibliotecas de terceiros.

7.3 IMPORT

Para usar uma classe do mesmo pacote, basta fazer referência a ela como foi feito até agora simplesmente escrevendo o próprio nome da classe. Se quisermos que a classe `Banco` fique dentro do pacote `br.com.caelum.contas`, ela deve ser declarada assim:

```
package br.com.caelum.contas;

class Banco {
    String nome;
}
```

Para a classe `Cliente` ficar no mesmo pacote, seguimos a mesma fórmula:

```
package br.com.caelum.contas;

class Cliente {
    String nome;
    String endereco;
}
```

A novidade chega ao tentar utilizar a classe `Banco` (ou `Cliente`) em uma outra classe que esteja fora desse pacote, por exemplo, no pacote `br.com.caelum.contas.main`:

```
package br.com.caelum.contas.main;

class TesteDoBanco {

    public static void main(String[] args) {
      br.com.caelum.contas.Banco meuBanco = new br.com.caelum.contas.Banco();
      meuBanco.nome = "Banco do Brasil";
      System.out.println(meuBanco.nome);
    }

}
```

Repare que precisamos referenciar a classe `Banco` com todo o nome do pacote na sua frente. Esse é o conhecido *Fully Qualified Name* de uma classe. Em outras palavras, esse é o verdadeiro nome de uma classe, por isso duas classes com o mesmo nome em pacotes diferentes não conflitam.

Mesmo assim, ao tentar compilar a classe anterior, surge um erro reclamando que a classe `Banco` não está visível.

Acontece que as classes só são visíveis para outras no **mesmo pacote** e, para permitir que a classe `TesteDoBanco` veja e acesse a classe `Banco` em outro pacote, precisamos alterar essa última e transformá-la em pública:

```
package br.com.caelum.contas;

public class Banco {
    String nome;
}
```

A palavra chave `public` libera o acesso para classes de outros pacotes. Do mesmo jeito que o compilador reclamou que a classe não estava visível, ele reclama que o atributo/variável membro também não está. É fácil deduzir como resolver o problema: utilizando novamente o modificador `public`:

```
package br.com.caelum.contas;

public class Banco {
    public String nome;
}
```

Podemos testar nosso exemplo anterior, lembrando que utilizar atributos como público não traz encapsulamento e está aqui como ilustração.

Voltando ao código do `TesteDoBanco`, é necessário escrever todo o pacote para identificar qual classe queremos usar? O exemplo que usamos ficou bem complicado de ler:

```
br.com.caelum.contas.Banco meuBanco = new br.com.caelum.contas.Banco();
```

Existe uma maneira mais simples de se referenciar a classe `Banco`: basta **importá-la** do pacote `br.com.caelum.contas`:

```
package br.com.caelum.contas.main;

// para podermos referenciar
// a Banco diretamente
import br.com.caelum.contas.Banco;

public class TesteDoBanco {

    public static void main(String[] args) {
        Banco meuBanco = new Banco();
        meuBanco.nome = "Banco do Brasil";
    }
```

```
}
```

Isso faz com que não precisemos nos referenciar utilizando o *fully qualified name*, podendo utilizar `Banco` dentro do nosso código em vez de escrever o longo `br.com.caelum.contas.Banco`.

> **PACKAGE, IMPORT, CLASS**
>
> É muito importante manter a ordem! Primeiro, aparece uma (ou nenhuma) vez o `package`; depois, pode aparecer um ou mais `import`s; e, por último, as declarações de classes.

> **IMPORT X.Y.Z.*;**
>
> É possível "importar um pacote inteiro" (todas as classes do pacote, **exceto os subpacotes**) através do coringa `*`:
>
> ```
> import java.util.*;
> ```
>
> Importar todas as classes de um pacote não implica em perda de performance em tempo de execução, mas pode trazer problemas com classes de mesmo nome! Além disso, importar de um em um é considerado boa prática, pois facilita a leitura para outros programadores. Uma IDE como o Eclipse já vai fazer isso por você, assim como a organização em diretórios.

7.4 ACESSO AOS ATRIBUTOS, CONSTRUTORES E MÉTODOS

Os modificadores de acesso existentes em Java são quatro, e até o momento já vimos três, mas só explicamos dois.

- `public` - Todos podem acessar aquilo que for definido como `public`. Classes, atributos, construtores e métodos podem ser `public`.
- `protected` - Aquilo que é `protected` pode ser acessado por todas as classes do mesmo pacote e por todas as classes que o estendam, mesmo que essas não estejam no mesmo pacote. Somente atributos, construtores e métodos podem ser `protected`.
- **padrão (sem nenhum modificador)** - Se nenhum modificador for utilizado, todas as classes do mesmo pacote têm acesso ao atributo, construtor, método ou classe.
- `private` - A única classe capaz de acessar os atributos, construtores e métodos privados é a própria classe. Classes, como conhecemos, não podem ser `private`, mas atributos, construtores e

métodos sim.

Classes públicas

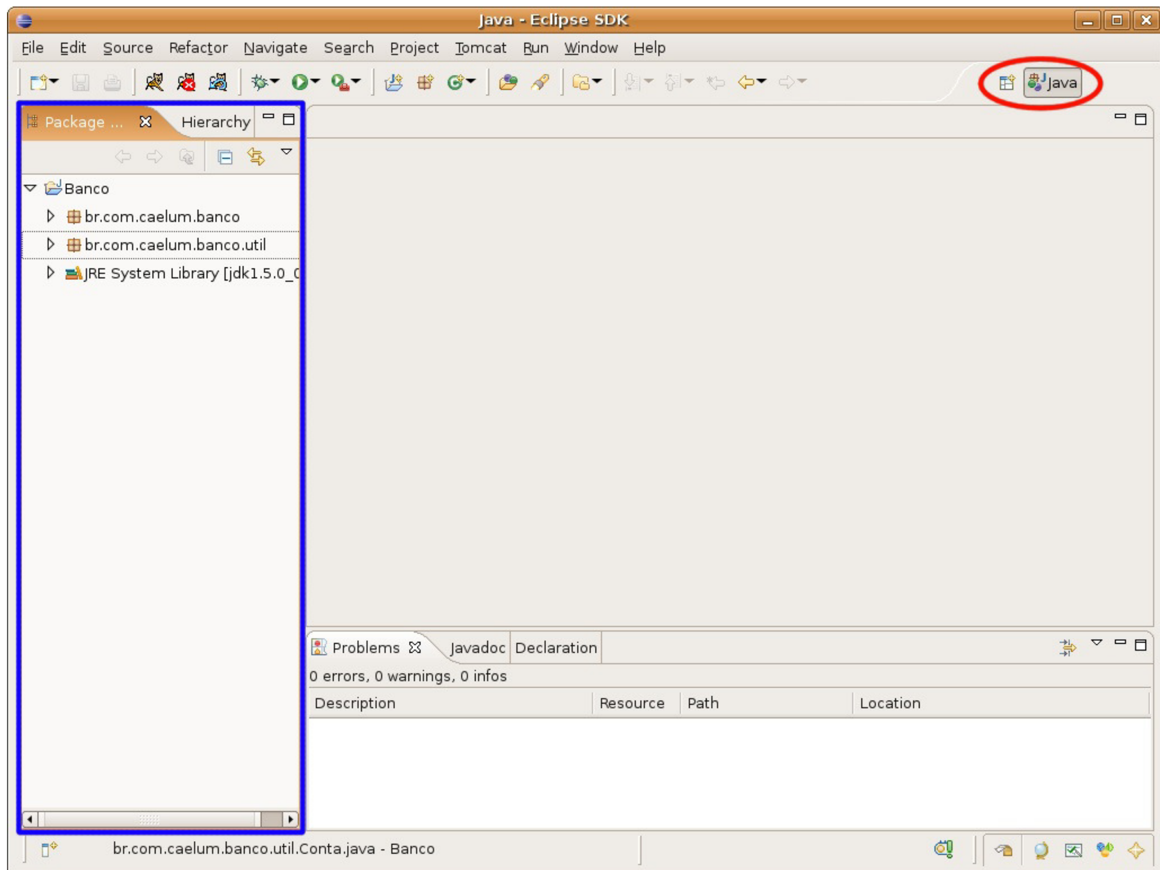
Para melhor organizar seu código, o Java não permite mais de uma classe pública por arquivo e o arquivo deve ser `NomeDaClasse.java` .

Uma vez que outros programadores irão utilizar essa classe, quando precisarem olhar o código da mesma, fica mais fácil encontrá-la sabendo que ela está no arquivo de mesmo nome.

Classes aninhadas podem ser `protected` ou `private` , mas esse é um tópico avançado que não será estudado nesse momento.

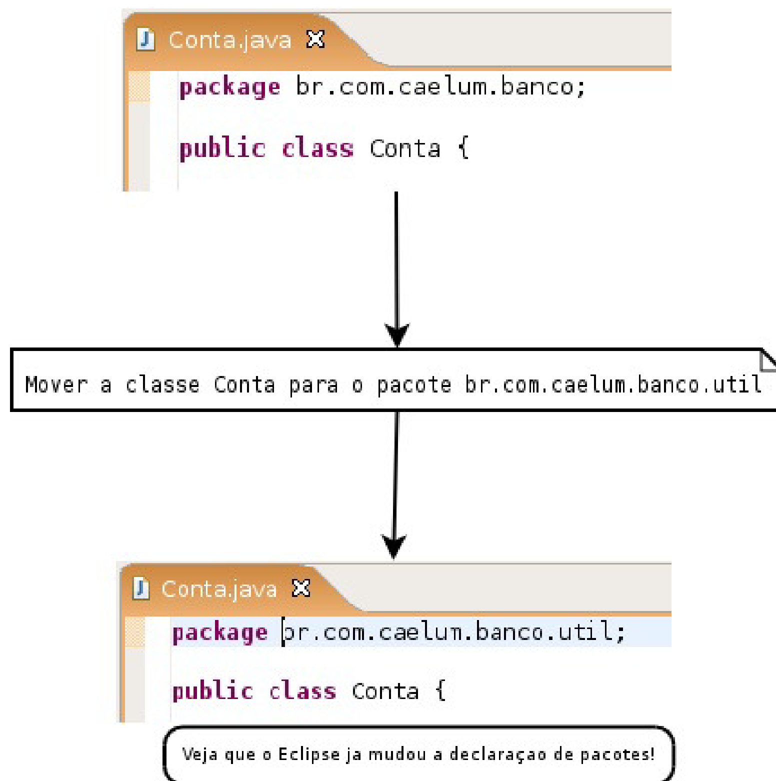
7.5 USANDO O ECLIPSE COM PACOTES

Você pode usar a perspectiva Java do Eclipse. A view principal de navegação é o *Package Explorer*, que agrupa classes pelos pacotes em vez de diretórios (você pode usá-la em conjunto com a *Navigator*, basta também abri-la pelo *Window/Show View/Package Explorer*).



Antes de movermos nossas classes, declare-as como públicas e coloque-as em seus respectivos arquivos: um arquivo para cada classe.

Você pode mover uma classe de pacote arrastando-a para o destino desejado. Repare que o Eclipse já declara `package` s e `import` s necessários:



No Eclipse nunca precisamos declarar um `import`, pois ele sempre vai sugerir isso quando usarmos o `Ctrl+Espaço` no nome de uma classe.

Você também pode usar o `Ctrl+1` no caso da declaração de pacote possuir algum erro.

7.6 EXERCÍCIOS: PACOTES

Atenção: utilize os recursos do Eclipse para realizar essas mudanças. Use a `view package-explorer`, que vai auxiliar bastante a manipulação dos arquivos e diretórios. Também utilize os quick fixes quando o Eclipse reclamar dos diversos problemas de compilação que aparecerão. É possível fazer esse exercício inteiro **sem modificar uma linha de código manualmente**. Aproveite para praticar e descobrir o Eclipse, evite usá-lo apenas como um editor de texto.

Por exemplo, com o Eclipse nunca precisamos nos preocupar com os imports: ao usar o auto complete, ele já joga o import lá em cima. E, se você não fez isso, ele sugere colocar o `import`.

1. Selecionando o `src` do seu projeto, faça **ctrl + N** e escreva `Package` para o seu sistema de Contas começar a utilizar pacotes. Na janela de criação de pacotes escreva o nome completo do pacote seguindo a convenção de código da Sun, desde o br, e o Eclipse tratará de fazer a separação das pastas corretamente. **Cuidado: para esse curso, os nomes dos pacotes precisam ser os seguintes:**

   - `br.com.caelum.contas.main` : colocar a classe com o método `main` aqui (o Teste)

- `br.com.caelum.contas.modelo` : colocar a classe `Conta`

  Antes de corrigir qualquer erro de compilação, primeiro **mova todas as suas classes**, sem deixar nenhuma no pacote *default*.

2. Se você ainda não tiver separado cada classe em um arquivo, essa é a hora de mudar isso. Coloque cada classe em seu respectivo arquivo `.java` . Faça isso independente de ela ser pública: é uma boa prática.

3. Caso o código não compile prontamente, repare que pelo menos algum método que declaramos é *package-private* quando, na verdade, precisamos que ele seja `public` . O mesmo vale para as classes: algumas delas **precisarão** ser públicas.

   Se houver algum erro de compilação, use o recurso de quick fix do Eclipse aqui: ele mesmo vai sugerir que o modificador de acesso deve ser público. Para isso, use o **ctrl + 1** em cada um dos erros, escolhendo o *quick fix* mais adequado para seu problema.

4. (Opcional) Abra a view `Navigator` para ver como ficou os arquivos no sistema de arquivos do seu sistema operacional. Para isso, use **ctrl + 3**, comece a digitar `Navigator` e escolha a opção de abrir essa view..

CAPÍTULO 8

FERRAMENTAS: JAR E JAVADOC

"Perder tempo em aprender coisas que não interessam, priva-nos de descobrir coisas interessantes" -- Carlos Drummond de Andrade

Ao término desse capítulo, você será capaz de:

- criar o JAR do seu aplicativo;
- colocar um JAR no build path do seu projeto;
- ler um javadoc;
- criar o javadoc do seu aplicativo.

8.1 ARQUIVOS, BIBLIOTECAS E VERSÕES

Assim que um programa fica pronto, é meio complicado enviar dezenas ou centenas de classes para cada cliente que quer utilizá-lo.

O jeito mais simples de trabalhar com um conjunto de classes é compactá-los em um arquivo só. O formato de compactação padrão é o **`ZIP`** com a extensão do arquivo compactado **`JAR`** .

> **O ARQUIVO .JAR**
>
> O arquivo **jar** ou **J**ava **AR**chive, possui um conjunto de classes (e arquivos de configurações) compactados, no estilo de um arquivo **`zip`** . O arquivo **`jar`** pode ser criado com qualquer compactador **`zip`** disponível no mercado, inclusive o programa **`jar`** que vem junto com o JDK.

Para criar um arquivo jar do nosso programa de banco, basta ir ao diretório onde estão contidas as classes e usar o comando a seguir para criar o arquivo **`banco.jar`** com todas as classes dos pacotes `br.com.caelum.util` e `br.com.caelum.banco` :

```
jar -cvf banco.jar br/com/caelum/util/*.class br/com/caelum/banco/*.class
```

Para usar esse arquivo `banco.jar` para rodar o `TesteDoBanco` basta rodar o `java` com o arquivo `jar` como argumento:

```
java -classpath banco.jar br.com.caelum.contas.main.TesteDoBanco
```

Para adicionar mais arquivos **`.jar`**, que podem ser bibliotecas, ao programa basta rodar o java da seguinte maneira:

```
java -classpath biblioteca1.jar;biblioteca2.jar NomeDaClasse
```

Vale lembrar que o ponto e vírgula utilizado só é válido em ambiente Windows. Em Linux, Mac e outros Unix, é o dois pontos (varia de acordo com o sistema operacional).

Há também um arquivo de manifesto que contém informações do seu jar como, por exemplo, qual classe ele vai rodar quando o jar for chamado. Mas não se preocupe pois, com o Eclipse, esse arquivo é gerado automaticamente.

BIBLIOTECAS

Diversas bibliotecas podem ser controladas de acordo com a versão por estarem sempre compactadas em um arquivo `.jar`. Basta verificar o nome da biblioteca (por exemplo `log4j-1.2.13.jar`) para descobrir a versão dela.

Então é possível rodar dois programas ao mesmo tempo, cada um utilizando uma versão da biblioteca através do parâmetro **`-classpath`** do java.

CRIANDO UM .JAR AUTOMATICAMENTE

Existem diversas ferramentas que servem para automatizar o processo de deploy, que consiste em compilar, gerar documentação, bibliotecas etc. As duas mais famosas são o **ANT** e o **MAVEN**, ambos são projetos do grupo Apache.

O Eclipse pode gerar facilmente um jar, porém, se o seu build é complexo e precisa preparar e copiar uma série de recursos, as ferramentas indicadas acima possuem sofisticadas maneiras de rodar um script batch.

Você pode também fazer o curso data dessa apostila na Caelum



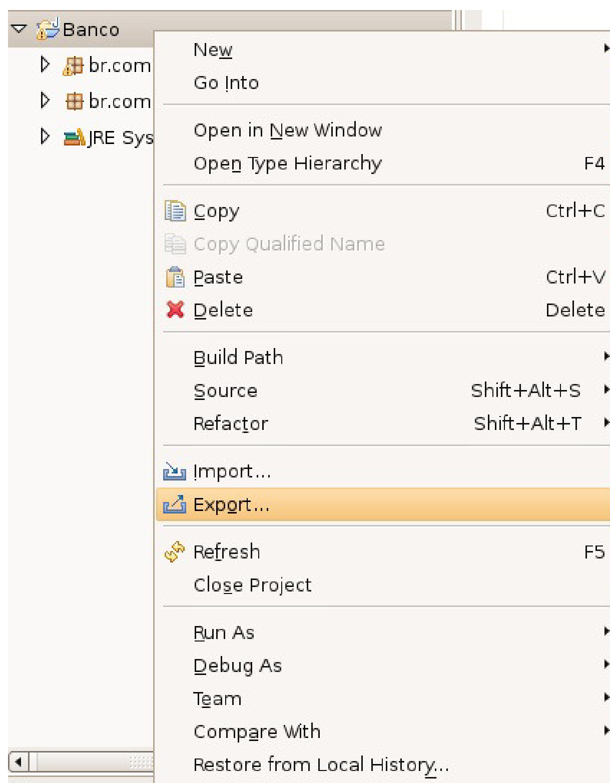
Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

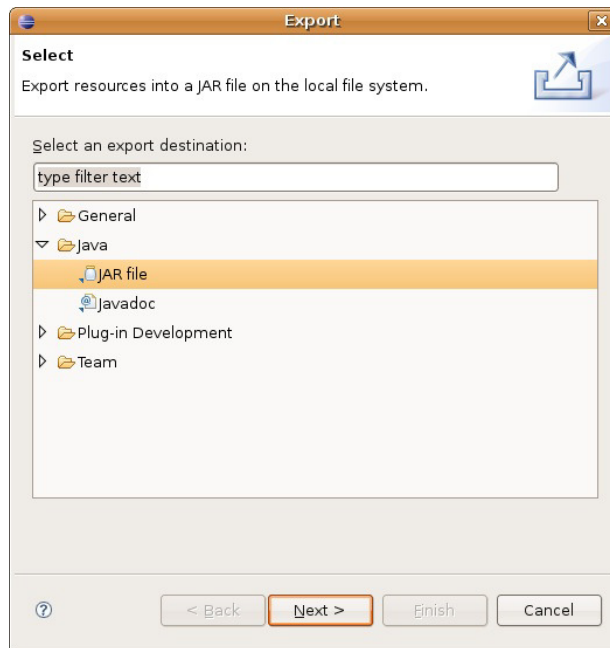
8.2 GERANDO O JAR PELO ECLIPSE

Neste exemplo, vamos gerar o arquivo JAR do nosso projeto a partir do Eclipse:

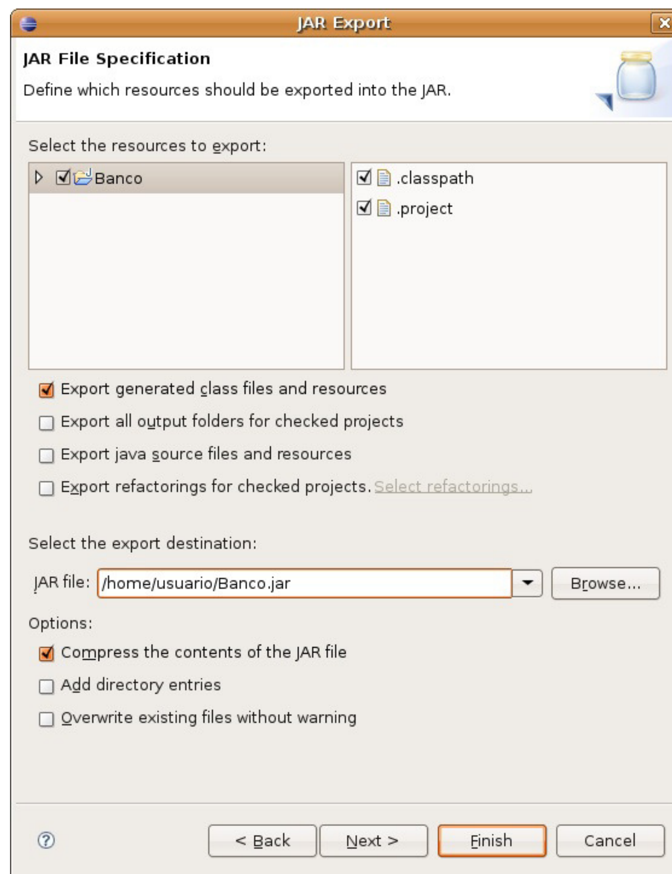
- Clique com o botão direito em cima do nome do seu projeto e selecione a opção Export.



- Na tela Export (como mostra a figura abaixo), selecione a opção "JAR file" e aperte o botão "Next".

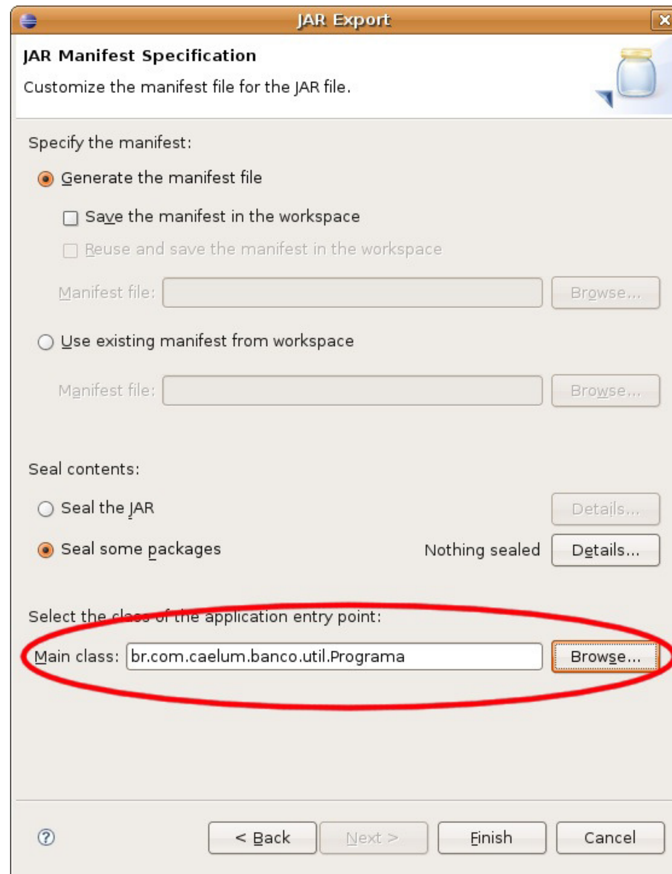


- Na opção "JAR file:", selecione o local que você deseja salvar o arquivo JAR. E aperte "Next".



- Na próxima tela, simplesmente clique em next, pois não há nenhuma configuração a ser feita.

- Na tela abaixo, na opção "select the class of the application entry point", você deve escolher qual classe será a classe que vai rodar automaticamente quando você executar o JAR.



- Entre na linha de comando: `java -jar banco.jar`

É comum dar um nome mais significativo aos JARs, incluindo nome da empresa, do projeto e versão, como `caelum-banco-1.0.jar` .

8.3 JAVADOC

Como vamos saber o que cada classe tem no Java? Quais são seus métodos, o que eles fazem?

E, a partir da Internet, você pode acessar através do link: http://download.java.net/jdk8/docs/api/index.html

No site da Oracle, você pode (e deve) baixar a documentação das bibliotecas do Java, frequentemente referida como "**javadoc**" ou API (sendo na verdade a documentação da API).

Java™ Platform
Standard Ed. 8

All Classes All Profiles

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font

All Classes

AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractAnnotationValueVisitor8
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractChronology
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContext
AbstractDocument.Content

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV NEXT FRAMES NO FRAMES

Java™ Platform, Standard Edition 8
API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

Profiles

- compact1
- compact2
- compact3

Packages

Package	Description
java.applet	Provides the classes neces with its applet context.

Nesta documentação, no quadro superior esquerdo, você encontra os pacotes e, no inferior esquerdo, está a listagem das classes e interfaces do respectivo pacote (ou de todos, caso nenhum tenha sido especificado). Clicando-se em uma classe ou interface, o quadro da direita passa a detalhar todos atributos e métodos.

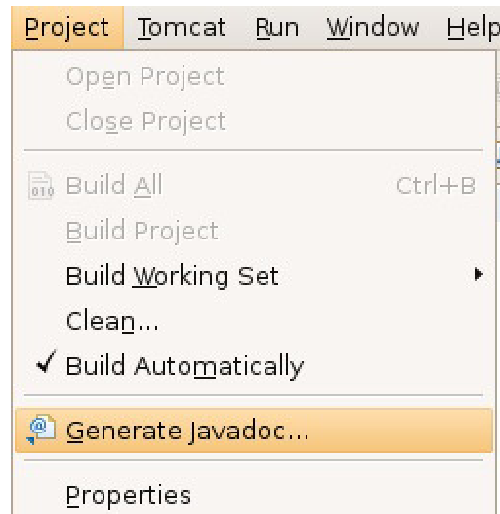
Repare que métodos e atributos privados não estão aí. O importante é documentar o que sua classe faz, e não como ela faz: detalhes de implementação, como atributos e métodos privados, não interessam ao desenvolvedor que usará a sua biblioteca (ou, ao menos, não deveriam interessar).

Você também consegue gerar esse javadoc a partir da linha de comando, com o comando: `javadoc` .

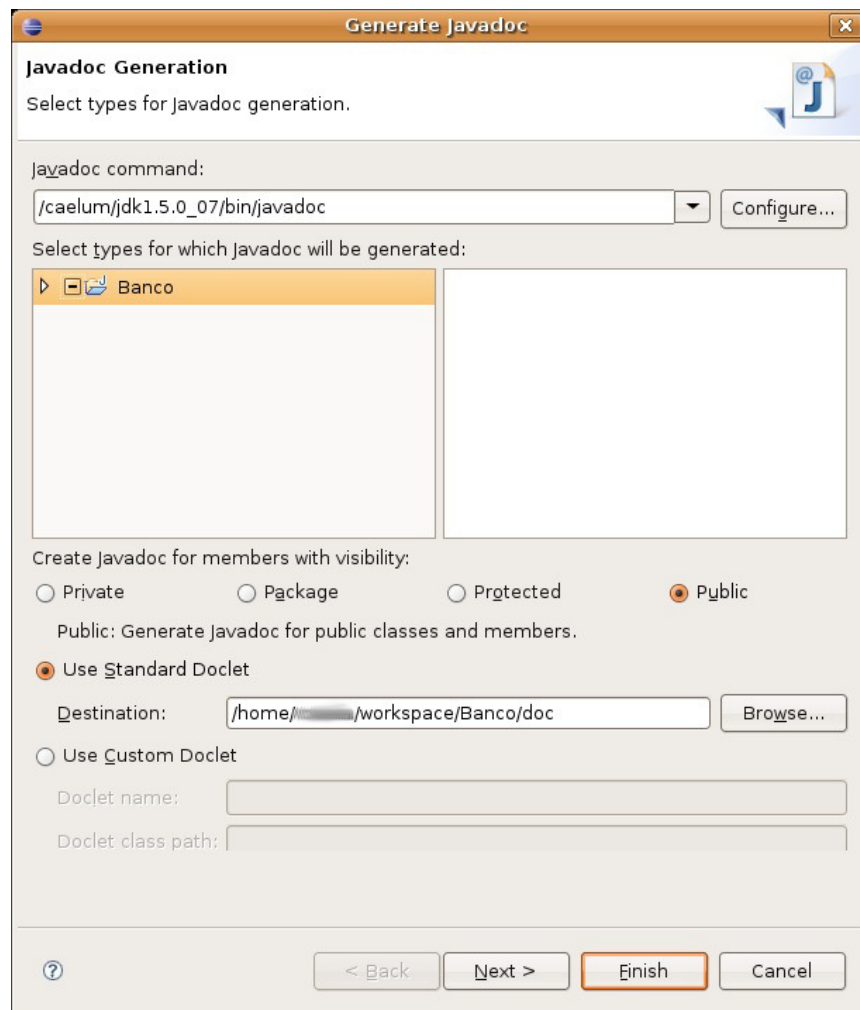
8.4 GERANDO O JAVADOC

Para gerar o Javadoc a partir do Eclipse é muito simples, siga os passos abaixo:

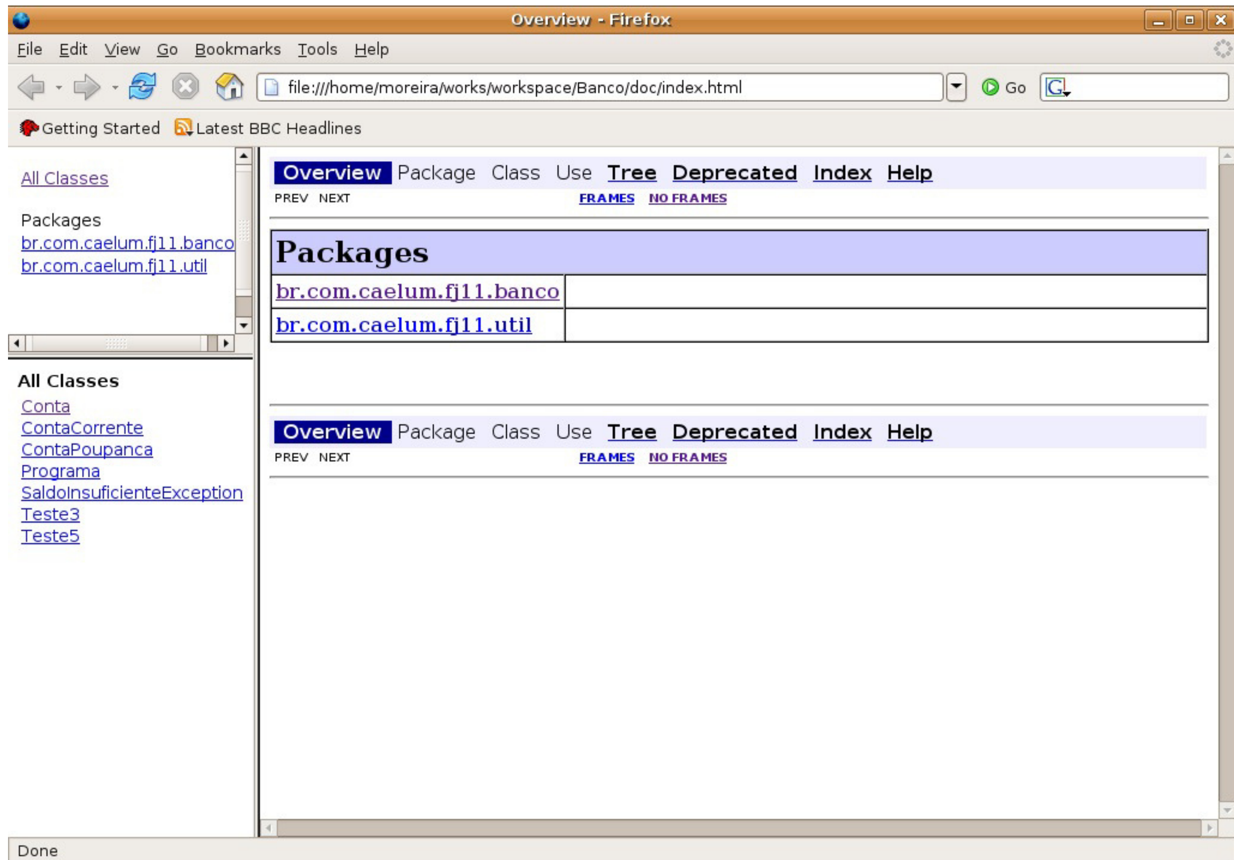
- Na barra de menu, selecione o menu Project, depois a opção "Generate Javadoc...". (apenas disponível se estiver na perspectiva Java, mas você pode acessar o mesmo wizard pelo export do projeto).



- Em seguida, aparecerão as opções para gerar a documentação do seu sistema, selecione todas as classes do seu sistema e deixe as outras opções como estão. Não esqueça de marcar o caminho da opção "Destination", pois é lá que estará sua documentação.



- Abra a documentação através do caminho que você marcou e abra o arquivo index.html, que vai chamar uma página semelhante a essa da figura abaixo.



Para colocarmos comentários na documentação, devemos adicionar ao código, sob forma de comentário, abrindo o texto com `/**` e fechando com `*/` e, nas outras linhas, apenas colocando `*` . Também podemos definir outras informações neste texto, como: autor, versão, parâmetros, retorno, etc. Adicione alguns comentários ao seu projeto como abaixo:

```
/**
 * Classe responsável por moldar as Contas do Banco
 *
 * @author Manoel Santos da Silva
 */

public class Conta{
    ...
}
```

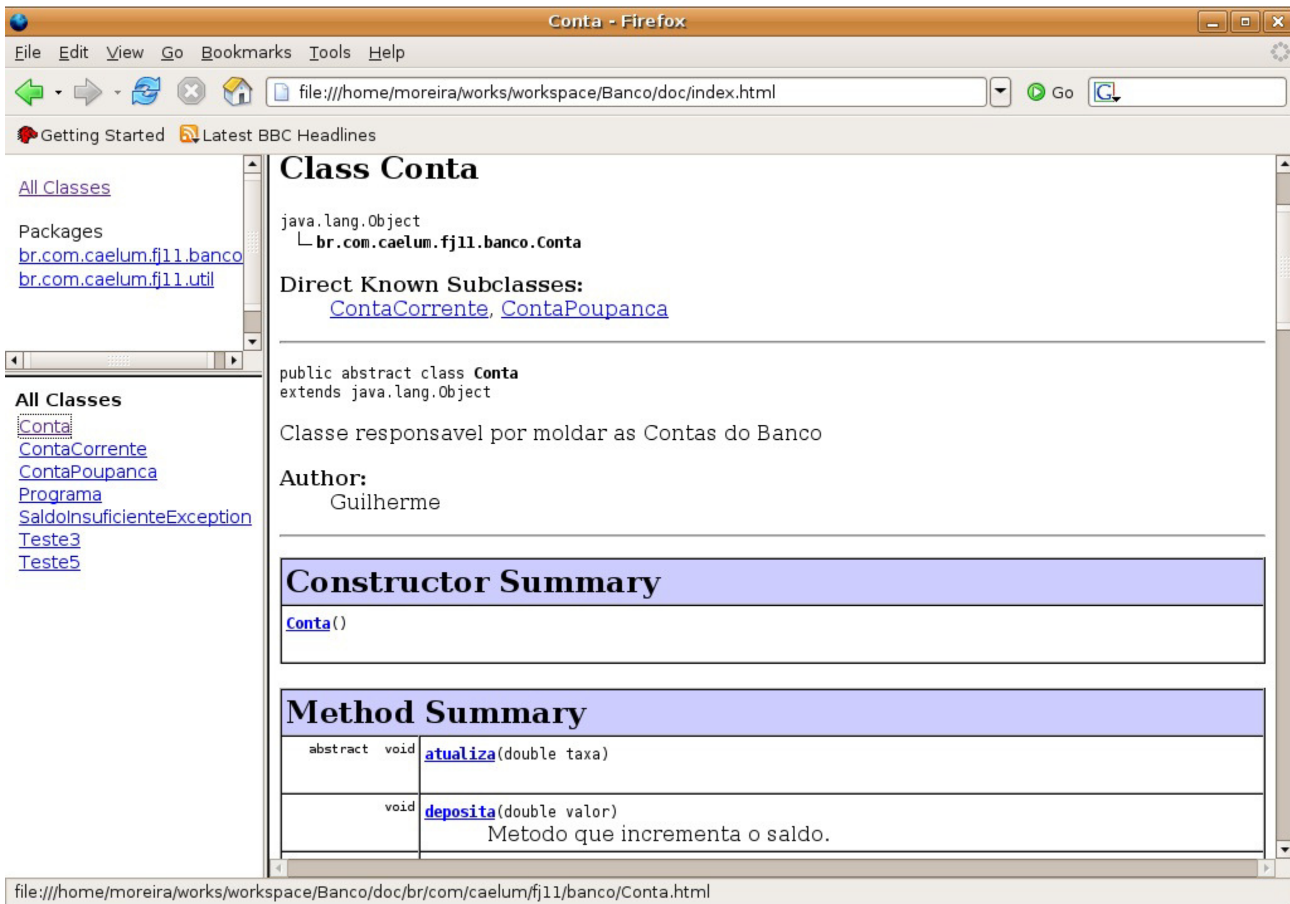
Ou adicione alguns comentários em algum método seu:

```
/**
 * Metodo que incrementa o saldo.
 * @param valor
 */

public void deposita(double valor) {
```

```
        ...
    }
```

Veja como ficou:



8.5 EXERCÍCIOS: JAR E JAVADOC

1. Gere um jar do seu sistema com o arquivo de manifesto. Execute-o com `java -jar` :

```
java -jar caelum-banco-1.0.jar
```

   Se o Windows ou o Linux foi configurado para trabalhar com a extensão *.jar*, basta você dar um duplo clique no arquivo, que ele será "executado": o arquivo `Manifest` será lido para que ele descubra qual é a classe com `main` que o Java deve processar.

2. Gere o Javadoc do seu sistema. Para isso, vá ao menu *Project*, depois à opção *Generate Javadoc*, se estiver na perspectiva Java. Se não, dê um clique com o botão direito no seu projeto, escolha *Export* e depois *javadoc* e siga o procedimento descrito na última seção deste capítulo.

   Independente da perspectiva que você usa no Eclipse, você também pode usar o **ctrl + 3** e começar a escrever `JavaDoc` , até que a opção de exportar o JavaDoc apareça.

INTERFACE VERSUS IMPLEMENTAÇÃO NOVAMENTE!

Repare que a documentação gerada não mostra o conteúdo dos métodos, nem atributos e métodos privados! Isso faz parte da implementação, e o que importa para quem usa uma biblioteca é a interface: o que ela faz.

8.6 IMPORTANDO UM JAR EXTERNO

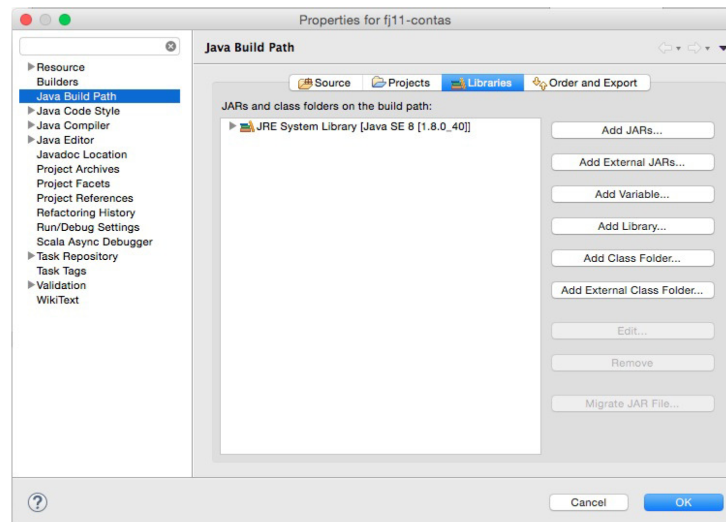
Já sabemos como documentar nosso projeto e gerar um jar para distribuí-lo mas ela ainda não tem uma interface gráfica do usuário. Se quisermos rodar o nosso sistema temos que executá-lo pelo terminal com os valores *hard-coded*. Seria mais interessante que tivéssemos uma interface mais amigável para que o usuário pudesse interagir com o nosso sistema. Ao mesmo tempo, não queremos nos preocupar nesse momento em criar todas as classes para representar essa interface gráfica, queremos apenas utilizar algo já pronto.

Para isso, vamos importar uma biblioteca externa. O próprio Eclipse já nos dá suporte para a importação de jars. Para fazer isso, basta ir no menu Project -> Properties, selecionar a opção Java Build Path, depois selecionar a aba Libraries e, finalmente, clicar no botão Add External Jars... . Agora é só selecionar o jar a ser importado e clicar em Open. Clique em Ok novamente para fechar a janela de importação e pronto! Nossa biblioteca já está pronta para ser utilizada.

8.7 EXERCÍCIOS: IMPORTANDO UM JAR

1. Vamos importar um jar que contém a interface gráfica do usuário para o nosso sistema de contas.

- Vá no menu **Project -> Properties**
- Selecione a opção **Java Build Path**
- Selecione a aba **Libraries**



- Clique no botão **Add External Jars...**
- Selecione o arquivo **fj11-lib-contas.jar** localizado na pasta dos arquivos dos cursos/11
- Clique no botão **Ok** para fechar a janela de importação

2. Para verificarmos que a importação deu certo, vamos chamar uma classe da biblioteca importada para exibir uma janela de boas-vindas.

   Crie uma classe `TestaJar` no pacote `br.com.caelum.contas.main` .

   Crie também o método `main` .

3. Dentro do método criado, vamos invocar o método `main` da classe `OlaMundo` que existe no jar importado. Seu código deve ficar dessa maneira:

```
package br.com.caelum.contas.main;

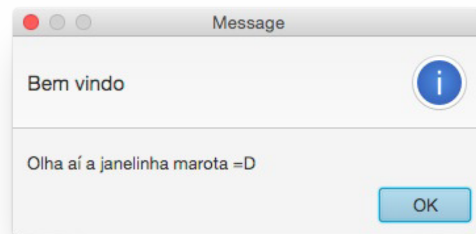
import br.com.caelum.javafx.api.main.OlaMundo;

public class TestaJar {

 public static void main(String[] args) {
     OlaMundo.main(args);
 }
}
```

   Não esqueça de importar a classe `OlaMundo` do pacote `br.com.caelum.javafx.api.main` . Use o atalho **ctrl + shift + O**.

4. Execute a sua aplicação e veja se apareceu uma janela de boas-vindas como a seguir:



8.8 MANIPULANDO A CONTA PELA INTERFACE GRÁFICA

Agora que já importamos o Jar que contém a interface gráfica, vamos dar uma olhada na primeira tela do nosso sistema:

Nesta tela, podemos perceber que temos botões para as ações de criação de conta, saque e depósito, que devem utilizar a implementação existente em nossa classe `Conta` .

Se quisermos visualizar a tela, podemos criar um `main` que chamará a classe `TelaDeContas` responsável pela sua exibição:

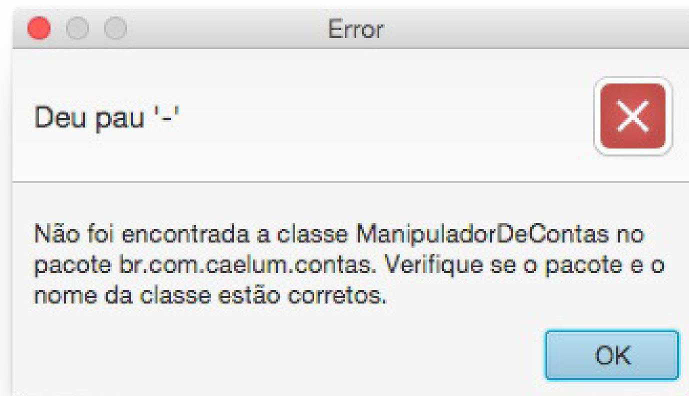
```
package br.com.caelum.contas.main;

import br.com.caelum.javafx.api.main.TelaDeContas;

public class TestaContas {

    public static void main(String[] args) {
        TelaDeContas.main(args);
    }
}
```

Ao executarmos a aplicação ocorrerá um erro:



Mas por que este erro ocorreu? Acontece que a tela precisa conhecer alguém que saiba executar as ações de saque e depósito na conta, que consiga buscar os dados da tela para popular a conta. Como não temos ninguém que saiba fazer isto ainda, ocorreu o erro.

Vamos então criar a classe `ManipuladorDeContas` que será responsável por fazer esta "ponte" entre a tela e a classe de `Conta` :

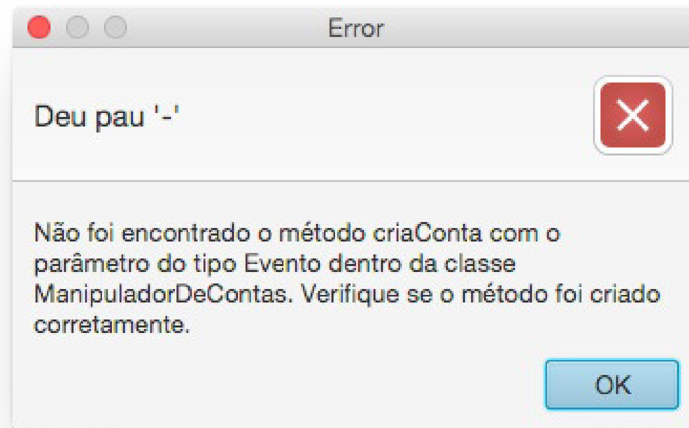
```
package br.com.caelum.contas;

public class ManipuladorDeContas {

}
```

Agora, ao executarmos a aplicação, veremos que a tela aparece com sucesso:

E se tentarmos clicar no botão de criação de conta? Também ocorre um erro!



Desta vez o erro indica que falta o método `criaConta` dentro da classe `ManipuladorDeContas`. Vamos então criá-lo:

```
public class ManipuladorDeContas {

    public void criaConta(){
        Conta conta = new Conta();
        conta.setAgencia("1234");
        conta.setNumero(56789);
        conta.setTitular("Batman");
    }
}
```

Para conseguirmos obter as informações da tela, todos os métodos que criaremos precisam receber um parâmetro do tipo `Evento` que conterá as informações digitadas. Mesmo que não utilizemos este parâmetro, precisamos recebê-lo.

```
import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeContas {

    public void criaConta(Evento evento){
        Conta conta = new Conta();
        conta.setAgencia("1234");
        conta.setNumero(56789);
        conta.setTitular("Batman");
    }
}
```

Se tentarmos executar a aplicação e clicar no botão **cria conta**, vemos que agora não ocorre mais nenhum erro mas ao mesmo tempo os dados da conta não são populados na tela. Isto acontece pois a variável `conta` é apenas local, ou seja, ela só existe dentro do método `criaConta`. Além disso se quiséssemos depositar um valor na conta, em qual conta depositaríamos? Ela não é visível para nenhum

outro método!

Precisamos que esta variável seja um atributo do `ManipuladorDeContas` . Vamos alterar:

```
import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeContas {

    private Conta conta;

    public void criaConta(Evento evento){
        this.conta = new Conta();
        this.conta.setAgencia("1234");
        this.conta.setNumero(56789);
        this.conta.setTitular("Batman");
    }
}
```

Testando agora conseguimos ver os dados da conta na tela!

Dados da conta

Agência: 1234

Número: 56789

Titular: Batman

Saldo: 0.0

Criar conta

Valor

Saca Deposita

Só falta criarmos os métodos `saca` e `deposita`. Vamos começar implementando o método `deposita`. Nele precisamos do valor digitado pelo usuário na tela e é pra isto que serve a classe `Evento`. Se quisermos buscar um valor do tipo `double`, podemos invocar o método `double` passando o nome do campo que queremos recuperar como parâmetro. Com o valor em mãos, podemos então passá-lo para o método desejado. Nosso método fica:

```
import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeContas {

    // ...

    public void deposita(Evento evento){
        double valorDigitado = evento.getDouble("valor");
        this.conta.deposita(valorDigitado);
    }
}
```

Podemos fazer o mesmo para o método `saca`:

```
import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeContas {

    // ...

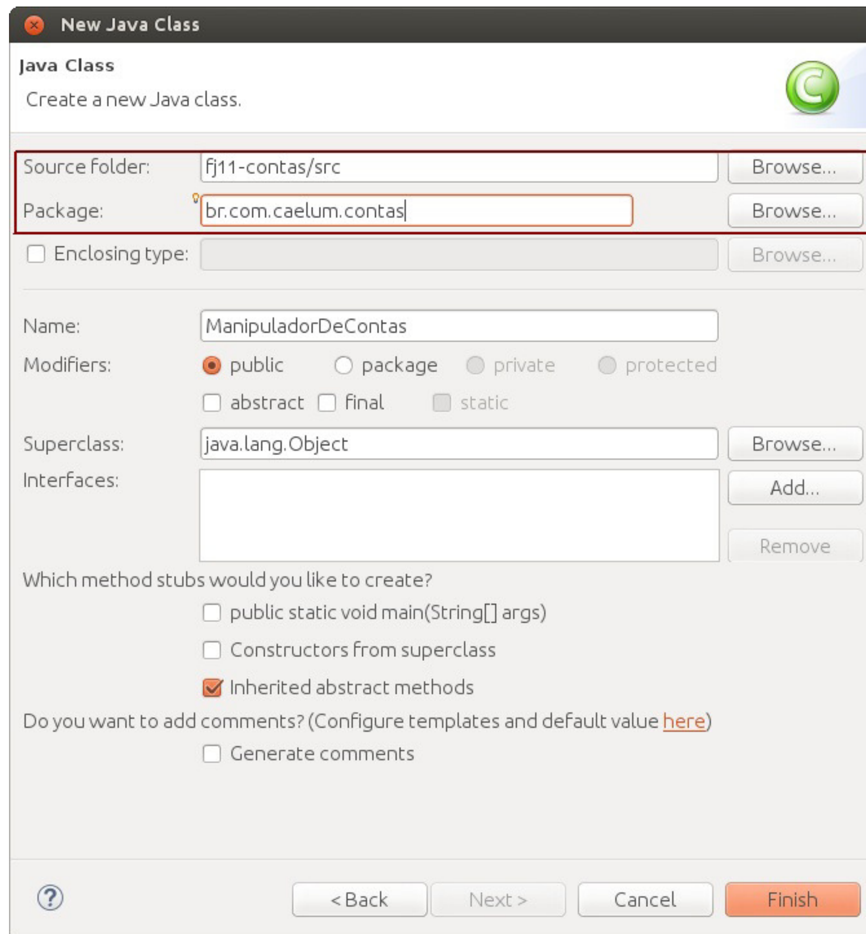
    public void deposita(Evento evento){
        double valorDigitado = evento.getDouble("valor");
        this.conta.deposita(valorDigitado);
    }

    public void saca(Evento evento){
        double valorDigitado = evento.getDouble("valor");
        this.conta.saca(valorDigitado);
    }
}
```

Agora conseguimos rodar a aplicação e chamar as ações de saque e depósito que o saldo é atualizado com sucesso!

8.9 EXERCÍCIOS: MOSTRANDO OS DADOS DA CONTA NA TELA

1. Crie a classe `ManipuladorDeContas` dentro do pacote `br.com.caelum.contas` . Repare que os pacotes `br.com.caelum.contas.main` e `br.com.caelum.contas.modelo` são subpacotes do pacote `br.com.caelum.contas` , portanto o pacote `br.com.caelum.contas` já existe. Para criar a classe neste pacote, basta selecioná-lo na janela de criação da classe:



A classe `ManipuladorDeContas` fará a ligação da `Conta` com a tela, por isso precisaremos declarar um atributo do tipo `Conta` .

2. Na classe `ManipuladorDeContas` , crie o método `criaConta` que recebe como parâmetro um objeto do tipo `Evento` . Instancie uma conta para o atributo `conta` e coloque os valores de `numero` , `agencia` e `titular` .

3. Com a conta instanciada, agora podemos implementar as funcionalidades de saque e depósito. Crie o método `deposita` que recebe um `Evento` , que é a classe que retorna os dados da tela nos tipos que precisamos. Por exemplo, se quisermos o valor a depositar sabemos que ele é do tipo `double` e que o nome do campo na tela é `valor` .

   Dica: a classe `Evento` tem o método `getDouble()` que retorna o conteúdo deste campo. Então, use `getDouble(“valor”)` quando precisar obter o conteúdo do campo `valor` .

1. Crie agora o método `saca` . Ele também deve receber um `Evento` nos mesmos moldes do `deposita` .

2. Precisamos agora testar nossa aplicação, crie a classe `TestaContas` dentro do pacote `br.com.caelum.contas` com um `main`. Nela vamos chamar o `main` da classe `TelaDeContas` que mostrará a tela de nosso sistema. Não se esqueça de fazer o import desta classe!

   Dica: para executar a tela da nossa aplicação, o método estático `main` da classe `TelaDeContas` deve ser invocado dentro do método `main` da classe `TestaContas` que você está criando agora.

   Rode a aplicação, crie a conta e tente fazer as operações de saque e depósito. Tudo deve funcionar normalmente.

CAPÍTULO 9

HERANÇA, REESCRITA E POLIMORFISMO

"O homem absurdo é aquele que nunca muda." -- Georges Clemenceau

Ao término desse capítulo, você será capaz de:

- dizer o que é herança e quando utilizá-la;
- reutilizar código escrito anteriormente;
- criar classes filhas e reescrever métodos;
- usar todo o poder que o polimorfismo dá.

9.1 REPETINDO CÓDIGO?

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe `Funcionario` :

```
public class Funcionario {
    private String nome;
    private String cpf;
    private double salario;
     // métodos devem vir aqui
}
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia:

```
public class Gerente {
    private String nome;
    private String cpf;
    private double salario;
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
```

```
    }

    // outros métodos
}
```

Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe `Funcionario` mais genérica, mantendo nela senha de acesso, e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios.

Essa é uma possibilidade, porém podemos começar a ter muito atributos opcionais, e a classe ficaria estranha. E em relação aos métodos? A classe `Gerente` tem o método `autentica` , que não faz sentido existir em um funcionário que não é gerente.

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

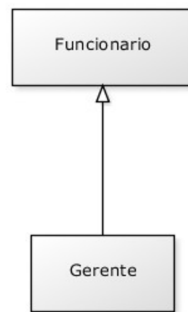
Existe um jeito, em Java, de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o `Gerente` tivesse tudo que um `Funcionario` tem, gostaríamos que ela fosse uma **extensão** de `Funcionario` . Fazemos isto através da palavra chave `extends` .

```
public class Gerente extends Funcionario {
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // setter da senha omitido
}
```

Em todo momento que criarmos um objeto do tipo `Gerente` , este objeto possuirá também os atributos definidos na classe `Funcionario` , pois um `Gerente` **é um** `Funcionario` :




```
public class TestaGerente {
    public static void main(String[] args) {
        Gerente gerente = new Gerente();

        // podemos chamar métodos do Funcionario:
        gerente.setNome("João da Silva");

        // e também métodos do Gerente!
        gerente.setSenha(4231);
    }
}
```

Dizemos que a classe `Gerente` **herda** todos os atributos e métodos da classe mãe, no nosso caso, a `Funcionario`. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente. Para acessar um membro privado na filha indiretamente, seria necessário que a mãe expusesse um outro método visível que invocasse esse atributo ou método privado.

Super e Sub classe

A nomenclatura mais encontrada é que `Funcionario` é a **superclasse** de `Gerente`, e `Gerente` é a **subclasse** de `Funcionario`. Dizemos também que todo `Gerente` **é um** `Funcionário`. Outra forma é dizer que `Funcionario` é classe **mãe** de `Gerente` e `Gerente` é classe **filha** de `Funcionario`.

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de `Funcionario`, `public`, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o `protected`, que fica entre o `private` e o `public`. Um atributo `protected` só pode ser acessado (visível) pela própria classe, por suas subclasses, e pelas classes que se encontram no mesmo pacote.

```
public class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;
    // métodos devem vir aqui
}
```

SEMPRE USAR PROTECTED?

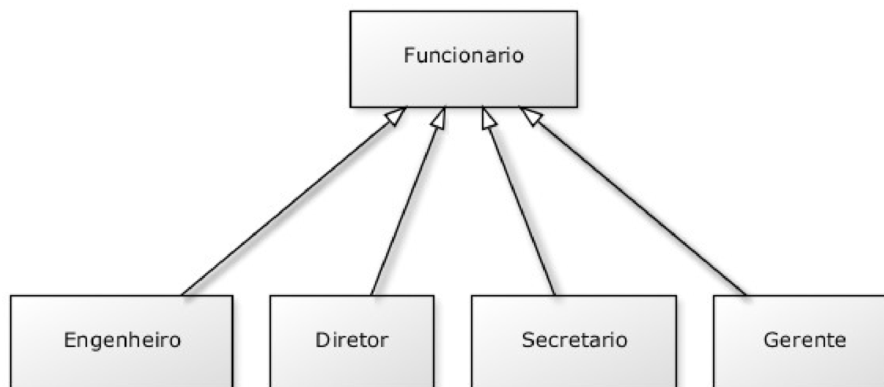
Então por que usar `private` ? Depois de um tempo programando orientado a objetos, você vai começar a sentir que nem sempre é uma boa ideia deixar que a classe filha acesse os atributos da classe mãe, pois isso quebra um pouco a ideia de que só aquela classe deveria manipular seus atributos. Essa é uma discussão um pouco mais avançada.

Além disso, não só as subclasses, mas também as outras classes que se encontram no mesmo pacote, podem acessar os atributos `protected` . Veja outras alternativas ao `protected` no exercício de discussão em sala de aula juntamente com o instrutor.

Da mesma maneira, podemos ter uma classe `Diretor` que estenda `Gerente` e a classe `Presidente` pode estender diretamente de `Funcionario` .

Fique claro que essa é uma decisão de negócio. Se Diretor vai estender de Gerente ou não, vai depender se, para você, Diretor *é um* Gerente.

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java.



Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

9.2 REESCRITA DE MÉTODO

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe `Funcionario` :

```
public class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 0.10;
    }
    // métodos
}
```

Se deixarmos a classe `Gerente` como ela está, ela vai herdar o método `getBonificacao` .

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso. Para consertar isso, uma das opções seria criar um novo método na classe `Gerente` , chamado, por exemplo, `getBonificacaoDoGerente` . O problema é que teríamos dois métodos em `Gerente` , confundindo bastante quem for usar essa classe, além de que cada um da uma resposta diferente.

No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos **reescrever** (reescrever, sobrescrever, *override*) este método:

```
public class Gerente extends Funcionario {
```

```
        int senha;
        int numeroDeFuncionariosGerenciados;

        public double getBonificacao() {
            return this.salario * 0.15;
        }
        // ...
    }
```

Agora o método está correto para o `Gerente`. Refaça o teste e veja que o valor impresso é o correto (750):

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

A ANOTAÇÃO @OVERRIDE

Há como deixar explícito no seu código que determinado método é a reescrita de um método da sua classe mãe. Fazemos isso colocando `@Override` em cima do método. Isso é chamado **anotação**. Existem diversas anotações e cada uma vai ter um efeito diferente sobre seu código.

```
@Override
public double getBonificacao() {
    return this.salario * 0.15;
}
```

Repare que, por questões de compatibilidade, isso não é obrigatório. Mas caso um método esteja anotado com @Override, ele necessariamente precisa estar reescrevendo um método da classe mãe.

9.3 INVOCANDO O MÉTODO REESCRITO

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe: realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um `Gerente` devemos fazer igual ao cálculo de um `Funcionario` porém adicionando R$ 1000. Poderíamos fazer assim:

```
public class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return this.salario * 0.10 + 1000;
    }
     // ...
}
```

Aqui teríamos um problema: o dia que o `getBonificacao` do `Funcionario` mudar, precisaremos mudar o método do `Gerente` para acompanhar a nova bonificação. Para evitar isso, o `getBonificacao` do `Gerente` pode chamar o do `Funcionario` utilizando a palavra chave `super`.

```
public class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public double getBonificacao() {
        return super.getBonificacao() + 1000;
    }
     // ...
}
```

Essa invocação vai procurar o método com o nome `getBonificacao` de uma super classe de `Gerente`. No caso ele logo vai encontrar esse método em `Funcionario`.

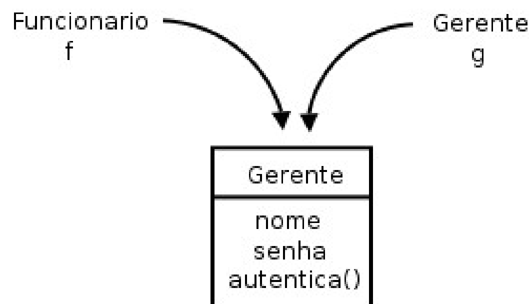
Essa é uma prática comum, pois muitos casos o método reescrito geralmente faz "algo a mais" que o método da classe mãe. Chamar ou não o método de cima é uma decisão sua e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

9.4 POLIMORFISMO

O que guarda uma variável do tipo `Funcionario`? Uma referência para um `Funcionario`, nunca o objeto em si.

Na herança, vimos que todo `Gerente` é um `Funcionario`, pois é uma extensão deste. Podemos nos referir a um `Gerente` como sendo um `Funcionario`. Se alguém precisa falar com um `Funcionario` do banco, pode falar com um `Gerente`! Porque? Pois `Gerente` **é um** `Funcionario`. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
```



Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.getBonificacao();
```

Qual é o retorno desse método? 500 ou 750? No Java, a invocação de método sempre vai ser **decidida em tempo de execução**. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo. Apesar de estarmos nos referenciando a esse `Gerente` como sendo um `Funcionario`, o método executado é o do `Gerente`. O retorno é 750.

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo `Funcionario`:

```
class ControleDeBonificacoes {
    private double totalDeBonificacoes = 0;

    public void registra(Funcionario funcionario) {
        this.totalDeBonificacoes += funcionario.getBonificacao();
    }

    public double getTotalDeBonificacoes() {
        return this.totalDeBonificacoes;
    }
}
```

E, em algum lugar da minha aplicação (ou no `main`, se for apenas para testes):

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();

Gerente funcionario1 = new Gerente();
funcionario1.setSalario(5000.0);
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.registra(funcionario2);

System.out.println(controle.getTotalDeBonificacoes());
```

Repare que conseguimos passar um `Gerente` para um método que recebe um `Funcionario` como argumento. Pense como numa porta na agência bancária com o seguinte aviso: "Permitida a entrada apenas de Funcionários". Um gerente pode passar nessa porta? Sim, pois `Gerente` **é um** `Funcionario`.

Qual será o valor resultante? Não importa que dentro do método registra do `ControleDeBonificacoes` receba `Funcionario`. Quando ele receber um objeto que realmente é um `Gerente`, o seu método reescrito será invocado. Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele**.

No dia em que criarmos uma classe `Secretaria`, por exemplo, que é filha de `Funcionario`,

precisaremos mudar a classe de `ControleDeBonificacoes` ? Não. Basta a classe `Secretaria` reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a reescrita de método: diminuir o acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou `ControleDeBonificacoes` pode nunca ter imaginado a criação da classe `Secretaria` ou `Engenheiro` . Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

HERANÇA *VERSUS* ACOPLAMENTO

Note que o uso de herança **aumenta** o acoplamento entre as classes, isto é, o quanto uma classe depende de outra. A relação entre classe mãe e filha é muito forte e isso acaba fazendo com que o programador das classes filhas tenha que conhecer a implementação da classe mãe e vice-versa - fica difícil fazer uma mudança pontual no sistema.

Por exemplo, imagine se tivermos que mudar algo na nossa classe `Funcionario` , mas não quiséssemos que todos os funcionários sofressem a mesma mudança. Precisaríamos passar por cada uma das filhas de `Funcionario` verificando se ela se comporta como deveria ou se devemos sobrescrever o tal método modificado.

Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde com a ajuda de Interfaces.

9.5 UM OUTRO EXEMPLO

Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
public class EmpregadoDaFaculdade {
    private String nome;
    private double salario;
    public double getGastos() {
        return this.salario;
    }
    public String getInfo() {
        return "nome: " + this.nome + " com salário " + this.salario;
    }
     // métodos de get, set e outros
}
```

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então? Reescrevemos o método. Assim como o `getGastos` é diferente, o `getInfo` também será, pois temos de mostrar as horas/aula também.

```
public class ProfessorDaFaculdade extends EmpregadoDaFaculdade {
    private int horasDeAula;
    public double getGastos() {
        return this.getSalario() + this.horasDeAula * 10;
    }
    public String getInfo() {
        String informacaoBasica = super.getInfo();
        String informacao = informacaoBasica + " horas de aula: "
                                                + this.horasDeAula;
        return informacao;
    }
    // métodos de get, set e outros que forem necessários
}
```

A novidade, aqui, é a palavra chave `super`. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.

Como tiramos proveito do polimorfismo? Imagine que temos uma classe de relatório:

```
public class GeradorDeRelatorio {
    public void adiciona(EmpregadoDaFaculdade f) {
        System.out.println(f.getInfo());
        System.out.println(f.getGastos());
    }
}
```

Podemos passar para nossa classe qualquer `EmpregadoDaFaculdade`! Vai funcionar tanto para professor, quanto para funcionário comum.

Um certo dia, muito depois de terminar essa classe de relatório, resolvemos aumentar nosso sistema, e colocar uma classe nova, que representa o `Reitor`. Como ele também é um `EmpregadoDaFaculdade`, será que vamos precisar alterar algo na nossa classe de `Relatorio`? Não. Essa é a ideia! Quem programou a classe `GeradorDeRelatorio` nunca imaginou que existiria uma

classe Reitor e, mesmo assim, o sistema funciona.

```
public class Reitor extends EmpregadoDaFaculdade {
     // informações extras
    public String getInfo() {
        return super.getInfo() + " e ele é um reitor";
    }
     // não sobrescrevemos o getGastos!!!
}
```

9.6 UM POUCO MAIS...

- Se não houvesse herança em Java, como você poderia reaproveitar o código de outra classe?
- Uma discussão muito atual é sobre o abuso no uso da herança. Algumas pessoas usam herança apenas para reaproveitar o código, quando poderiam ter feito uma **composição**. Procure sobre herança versus composição.
- Mesmo depois de reescrever um método da classe mãe, a classe filha ainda pode acessar o método antigo. Isto é feito através da palavra chave `super.método()` . Algo parecido ocorre entre os construtores das classes, o que?

MAIS SOBRE O MAU USO DA HERANÇA

No blog da Caelum existe um artigo interessante abordando esse tópico:

http://blog.caelum.com.br/2006/10/14/como-nao-aprender-orientacao-a-objetos-heranca/

James Gosling, um dos criadores do Java, é um crítico do mau uso da herança. Nesta entrevista ele discute a possibilidade de se utilizar apenas interfaces e composição, eliminando a necessidade da herança:

http://www.artima.com/intv/gosling3P.html

9.7 EXERCÍCIOS: HERANÇA E POLIMORFISMO

1. Vamos ter mais de um tipo de conta no nosso sistema então vamos precisar de uma nova tela para cadastrar os diferentes tipos de conta. Essa tela já está pronta e para utilizá-la só precisamos alterar a classe que estamos chamando no método `main()` no `TestaContas.java` :

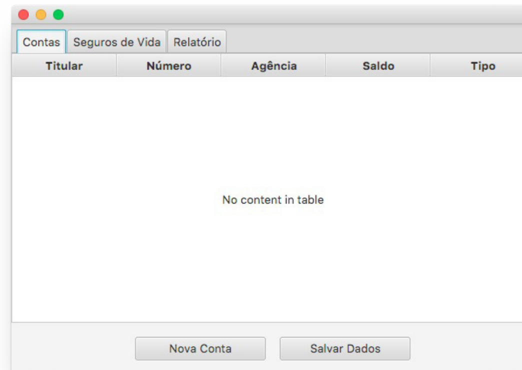
```
package br.com.caelum.contas.main;

import br.com.caelum.javafx.api.main.SistemaBancario;

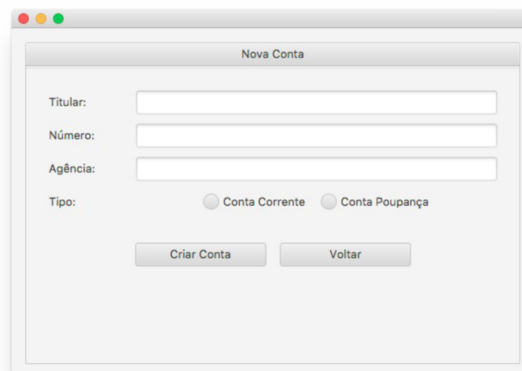
public class TestaContas {
```

```
    public static void main(String[] args) {
        SistemaBancario.mostraTela(false);
        // TelaDeContas.main(args);
    }
}
```

2. Ao rodar a classe `TestaContas` agora, teremos a tela abaixo:



Vamos entrar na tela de criação de contas para vermos o que precisamos implementar para que o sistema funcione. Para isso, clique no botão **Nova Conta**. A seguinte tela aparecerá:



Podemos perceber que além das informações que já tínhamos na conta, temos agora o tipo: se queremos uma conta corrente ou uma conta poupança. Vamos então criar as classes correspondentes.

- Crie a classe `ContaCorrente` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`
- Crie a classe `ContaPoupanca` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`

3. Precisamos pegar os dados da tela para conseguirmos criar a conta correspondente. Na classe `ManipuladorDeContas` vamos alterar o método `criaConta`. Atualmente, apenas criamos uma nova conta com os dados direto no código. Vamos fazer com que agora os dados sejam recuperados da tela para colocarmos na nova conta. Para fazer isso utililize o objeto do tipo `evento` que é exigido como parâmetro do método `criaConta`.

   Dicas: A seguir algumas informações importantes sobre a classe `Evento` (a qual é responsável pela tela Nova Conta):

   • Para obter o conteúdo do campo `agencia`, invoque o método `getString("agencia");`

   • Para obter o conteúdo do campo `numero`, invoque o método `getInt("numero"));`

   • Para obter o conteúdo do campo `titular`, invoque o método `evento.getString("titular");`

   • Observe na figura anterior que agora precisamos escolher que tipo de conta queremos criar (Conta Corrente ou Conta Poupança) e portanto deveremos recuperar o tipo da conta escolhido e criar a conta correspondente.

   Para isso, ao invés de criar um objeto do tipo 'Conta', vamos usar o método `getSelecionadoNoRadio` do objeto `evento` para pegar o tipo, fazer um `if` para verificar esse tipo e só depois criar o objeto do tipo correspondente. A seguir um trecho do código:

```
public void criaConta(Evento evento){
    String tipo = evento.getSelecionadoNoRadio("tipo");
    if (tipo.equals("Conta Corrente")) {
    //complete o código

}
```

4. Apesar de já conseguirmos criar os dois tipos de contas, nossa lista não consegue exibir o tipo de cada conta na lista da tela inicial. Para resolver isso, podemos criar um método `getTipo` em cada uma das classes que representam nossas contas fazendo com que a conta corrente devolva a string "Conta Corrente" e a conta poupança devolva a string "Conta Poupança".

5. **Atenção!** Altere os métodos `saca` e `deposita` para buscarem o campo `valorOperacao` ao invés de apenas `valor` na classe `ManipuladorDeContas`.

6. Vamos mudar o comportamento da operação de saque de acordo com o tipo de conta que estiver sendo utilizada. Na classe `ManipuladorDeContas` vamos alterar o método `saca` para tirar 10 centavos de cada saque em uma conta corrente. A seguir um trecho do código:

```
public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    if (this.conta.getTipo().equals("Conta Corrente")){
   //complete o código

}
```

Dica: ao tentarmos chamar o método `getTipo` , o Eclipse reclamou que esse método não existe na classe `Conta` apesar de existir nas classes filhas. O que fazer para resolver isto?

7. O código compila mas temos um outro problema. A lógica do nosso saque vazou para a classe `ManipuladorDeContas` . Se algum dia precisarmos alterar o valor da taxa no saque, teríamos que mudar em todos os lugares onde fazemos uso do método `saca` . Esta lógica deveria estar encapsulada dentro do método `saca` de cada conta. Como resolver isso?

   Dica: repare que, para acessar o atributo `saldo` herdado da classe `Conta` , **você vai precisar mudar o modificador de visibilidade de saldo para `protected`** .

   Agora que a lógica está encapsulada, você precisa corrigir o método `saca` da classe `ManipuladorDeContas` .

   Perceba que agora tratamos a conta de forma genérica!

8. Rode a classe `TestaContas` , adicione uma conta de cada tipo e veja se o tipo é apresentado corretamente na lista de contas da tela inicial.

   Agora, clique na conta corrente apresentada na lista para abrir a tela de detalhes de contas. Teste as operações de saque e depósito e perceba que a conta apresenta o comportamento de uma conta corrente conforme o esperado.

   E se tentarmos realizar uma transferência da conta corrente para a conta poupança? O que acontece?

9. Agora você precisa implementar o método `transfere` na classe `Conta` e na classe `ManipuladorDeContas` . Para isso, observe o seguinte:

   • O método `transfere` da classe `Conta` , deve receber como parâmetro, duas variáveis, uma para o valor a ser transferido e outra para a conta de destino.

   • O método `transfere` da classe `ManipuladorDeContas` deve existir para fazer o vínculo entre a tela e a classe `Conta` , assim ele deve receber um objeto do tipo `Evento` como parâmetro.

   • No corpo do método, por meio do objeto do tipo `Evento` , deve-se invocar o método `getSelecionadoNoCombo("destino");`

   • Em seguida, por meio do objeto do tipo `Conta` , deve-se chamar o método `transfere` da classe `Conta` para que a transferência seja de fato realizada.

   Rode de novo a aplicação e teste a operação de transferência.

10. Considere o código abaixo:

```
Conta c = new Conta();
```

```
ContaCorrente cc = new ContaCorrente();
ContaPoupanca cp = new ContaPoupanca();
```

Se mudarmos esse código para:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual é a utilidade disso? Realmente, essa não é a maneira mais útil do polimorfismo. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é, como fazemos na classe `ManipuladorDeContas` .

É **extremamente importante** perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto, não importando como nos referimos a ele.

11. (Opcional) A nossa classe `Conta` devolve a palavra "Conta" no método `getTipo` . Use a palavra chave `super` nos métodos `getTipo` reescritos nas classes filhas, para não ter de reescreve a palavra "Conta" ao devolver os tipos "Conta Corrente" e "Conta Poupança".

12. (Opcional) Se você precisasse criar uma classe `ContaInvestimento` , e seu método `saca` fosse complicadíssimo, você precisaria alterar a classe `ManipuladorDeContas` ?

9.8 DISCUSSÕES EM AULA: ALTERNATIVAS AO ATRIBUTO PROTECTED

Discuta com o instrutor e seus colegas alternativas ao uso do atributo `protected` na herança. Preciso realmente afrouxar o encapsulamento do atributo por causa da herança? Como fazer para o atributo continuar `private` na mãe e as filhas conseguirem de alguma formar trabalhar com ele?

CAPÍTULO 10

CLASSES ABSTRATAS

"Dá-se importância aos antepassados quando já não temos nenhum." -- François Chateaubriand

Ao término desse capítulo, você será capaz de utilizar classes abstratas, quando necessário.

10.1 REPETINDO MAIS CÓDIGO?

Vamos recordar em como pode estar nossa classe `Funcionario` :

```
public class Funcionario {

    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros métodos aqui

}
```

Considere o nosso `ControleDeBonificacao` :

```
public class ControleDeBonificacoes {

    private double totalDeBonificacoes = 0;

        public void registra(Funcionario f) {
            System.out.println("Adicionando bonificação do funcionario: " + f);
            this.totalDeBonificacoes += f.getBonificacao();
        }

        public double getTotalDeBonificacoes() {
            return this.totalDeBonificacoes;
        }
}
```

Nosso método `registra` recebe qualquer referência do tipo `Funcionario` , isto é, podem ser objetos do tipo `Funcionario` e qualquer de seus subtipos: `Gerente` , `Diretor` e, eventualmente, alguma nova subclasse que venha ser escrita, sem prévio conhecimento do autor da `ControleDeBonificacao` .

Estamos utilizando aqui a classe `Funcionario` para o polimorfismo. Se não fosse ela, teríamos um

grande prejuízo: precisaríamos criar um método `registra` para receber cada um dos tipos de `Funcionario`, um para `Gerente`, um para `Diretor`, etc. Repare que perder esse poder é muito pior do que a pequena vantagem que a herança traz em herdar código.

Porém, em alguns sistemas, como é o nosso caso, usamos uma classe com apenas esses intuitos: de economizar um pouco código e ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.

Faz sentido ter uma referência do tipo `Funcionario`? Essa pergunta é diferente de saber se faz sentido ter um objeto do tipo `Funcionario`: nesse caso, faz sim e é muito útil.

Referenciando `Funcionario` temos o polimorfismo de referência, já que podemos receber qualquer objeto que seja um `Funcionario`. Porém, dar `new` em `Funcionario` pode não fazer sentido, isto é, não queremos receber um objeto do tipo `Funcionario`, mas sim que aquela referência seja ou um `Gerente`, ou um `Diretor`, etc. Algo mais **concreto** que um `Funcionario`.

```
ControleDeBonificacoes cdb = new ControleDeBonificacoes();
Funcionario f = new Funcionario();
cdb.adiciona(f); // faz sentido?
```

Vejamos um outro caso em que não faz sentido ter um objeto daquele tipo, apesar da classe existir: imagine a classe `Pessoa` e duas filhas, `PessoaFisica` e `PessoaJuridica`. Quando puxamos um relatório de nossos clientes (uma array de `Pessoa` por exemplo), queremos que cada um deles seja ou uma `PessoaFisica`, ou uma `PessoaJuridica`. A classe `Pessoa`, nesse caso, estaria sendo usada apenas para ganhar o polimorfismo e herdar algumas coisas: não faz sentido permitir instanciá-la.

Para resolver esses problemas, temos as classes abstratas.

10.2 CLASSE ABSTRATA

O que, exatamente, vem a ser a nossa classe `Funcionario` ? Nossa empresa tem apenas `Diretores` , `Gerentes` , `Secretárias` , etc. Ela é uma classe que apenas idealiza um tipo, define apenas um rascunho.

Para o nosso sistema, é inadmissível que um objeto seja apenas do tipo `Funcionario` (pode existir um sistema em que faça sentido ter objetos do tipo `Funcionario` ou apenas `Pessoa` , mas, no nosso caso, não).

Usamos a palavra chave `abstract` para impedir que ela possa ser instanciada. Esse é o efeito direto de se usar o modificador `abstract` na declaração de uma classe:

```
public abstract class Funcionario {

    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros atributos e métodos comuns a todos Funcionarios

}
```

E, no meio de um código:

```
Funcionario f = new Funcionario(); // não compila!!!
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Cannot instantiate the type Funcionario

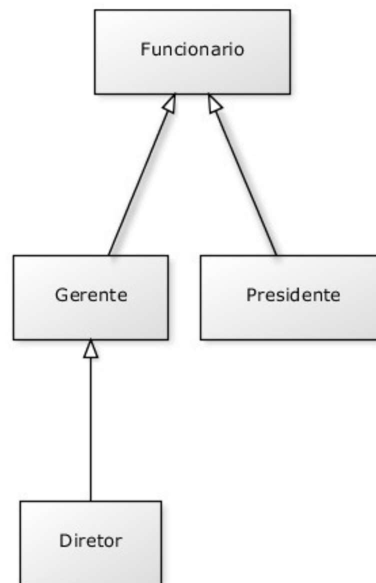
        at br.com.caelum.empresa.TestaFuncionario.main(TestaFuncionario.java:5)
```

O código acima não compila. O problema é instanciar a classe - criar referência, você pode. Se ela não pode ser instanciada, para que serve? Serve para o polimorfismo e herança dos atributos e métodos, que são recursos muito poderosos, como já vimos.

Vamos então herdar dessa classe, reescrevendo o método `getBonificacao` :

```
public class Gerente extends Funcionario {

    public double getBonificacao() {
        return this.salario * 1.4 + 1000;
    }
}
```



Mas qual é a real vantagem de uma classe abstrata? Poderíamos ter feito isto com uma herança comum. Por enquanto, a única diferença é que não podemos instanciar um objeto do tipo `Funcionario` , que já é de grande valia, dando mais consistência ao sistema.

Fique claro que a nossa decisão de transformar `Funcionario` em uma classe abstrata dependeu do nosso domínio. Pode ser que, em um sistema com classes similares, faça sentido que uma classe análoga a `Funcionario` seja concreta.

10.3 MÉTODOS ABSTRATOS

Se o método `getBonificacao` não fosse reescrito, ele seria herdado da classe mãe, fazendo com que devolvesse o salário mais 20%.

Levando em consideração que cada funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado, faz algum sentido ter esse método na classe `Funcionario` ? Será que existe uma bonificação padrão para todo tipo de `Funcionario` ? Parece que não, cada classe filha terá um método diferente de bonificação pois, de acordo com nosso sistema, não existe uma regra geral: queremos que cada pessoa que escreve a classe de um `Funcionario` diferente (subclasses de `Funcionario`) reescreva o método `getBonificacao` de acordo com as suas regras.

Poderíamos, então, jogar fora esse método da classe `Funcionario` ? O problema é que, se ele não existisse, não poderíamos chamar o método apenas com uma referência a um `Funcionario` , pois ninguém garante que essa referência aponta para um objeto que possui esse método. Será que então devemos retornar um código, como um número negativo? Isso não resolve o problema: se esquecermos de reescrever esse método, teremos dados errados sendo utilizados como bônus.

Existe um recurso em Java que, em uma classe abstrata, podemos escrever que determinado método será **sempre** escrito pelas classes filhas. Isto é, um **método abstrato**.

Ele indica que todas as classes filhas (concretas, isto é, que não forem abstratas) devem reescrever esse método ou não compilarão. É como se você herdasse a responsabilidade de ter aquele método.

COMO DECLARAR UM MÉTODO ABSTRATO

Às vezes, não fica claro como declarar um método abstrato.

Basta escrever a palavra chave `abstract` na assinatura do mesmo e colocar um ponto e vírgula em vez de abre e fecha chaves!

```
public abstract class Funcionario {

    public abstract double getBonificacao();

    // outros atributos e métodos

}
```

Repare que não colocamos o corpo do método e usamos a palavra chave `abstract` para definir o mesmo. Por que não colocar corpo algum? Porque esse método nunca vai ser chamado, sempre que alguém chamar o método `getBonificacao`, vai cair em uma das suas filhas, que realmente escreveram o método.

Qualquer classe que estender a classe `Funcionario` será obrigada a reescrever este método, tornando-o "concreto". Se não reescreverem esse método, um erro de compilação ocorrerá.

O método do `ControleDeBonificacao` estava assim:

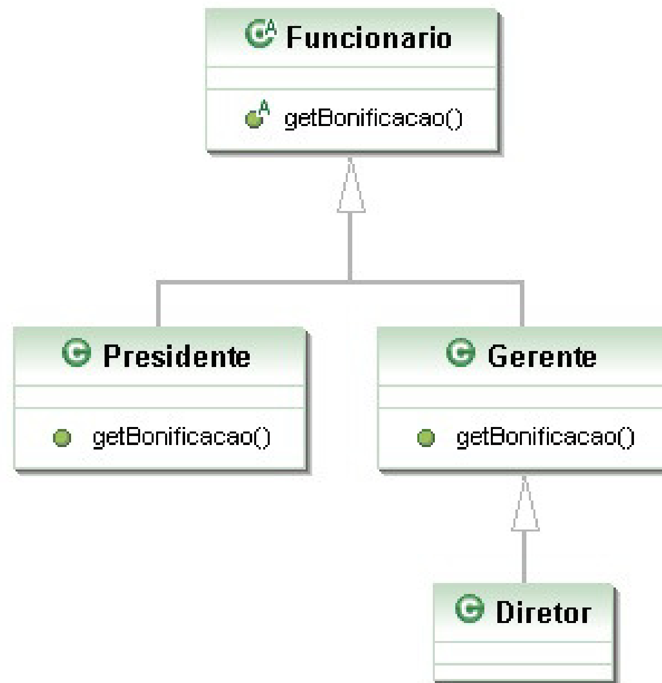
```
public void registra(Funcionario f) {
    System.out.println("Adicionando bonificação do funcionario: " + f);
    this.totalDeBonificacoes += f.getBonificacao();
}
```

Como posso acessar o método `getBonificacao` se ele não existe na classe `Funcionario`?

Já que o método é abstrato, **com certeza** suas subclasses têm esse método, o que garante que essa invocação de método não vai falhar. Basta pensar que uma referência do tipo `Funcionario` nunca aponta para um objeto que não tem o método `getBonificacao`, pois não é possível instanciar uma classe abstrata, apenas as concretas. Um método abstrato obriga a classe em que ele se encontra ser abstrata, o que garante a coerência do código acima compilar.

10.4 AUMENTANDO O EXEMPLO

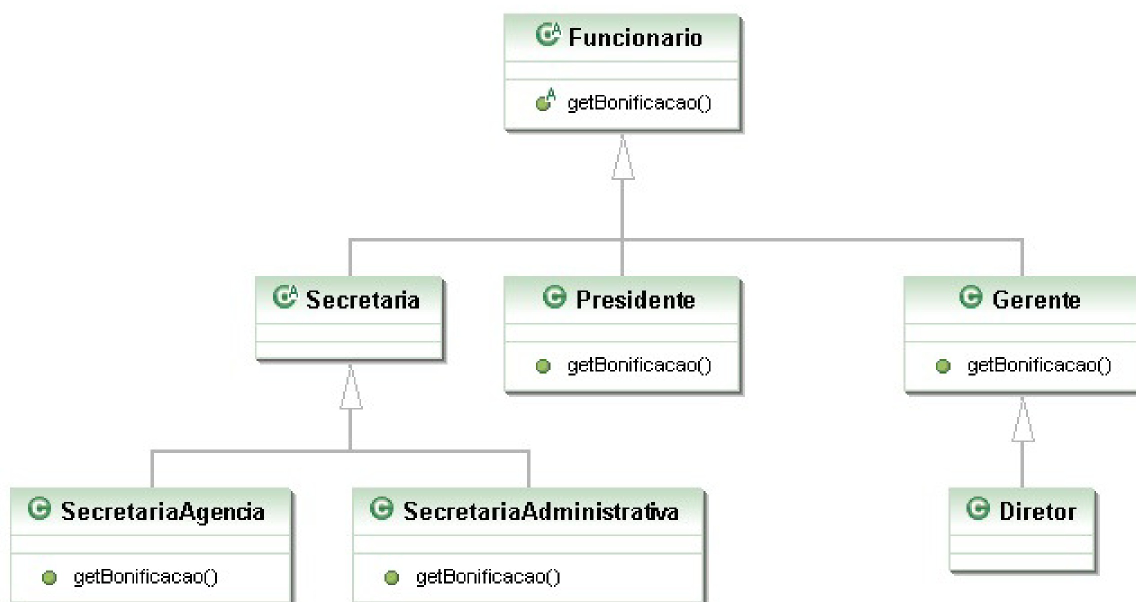
E se, no nosso exemplo de empresa, tivéssemos o seguinte diagrama de classes com os seguintes métodos:



Ou seja, tenho a classe abstrata `Funcionario` , com o método abstrato `getBonificacao` ; as classes `Gerente` e `Presidente` estendendo `Funcionario` e implementando o método `getBonificacao` ; e, por fim, a classe `Diretor` , que estende `Gerente` , mas não implementa o método `getBonificacao` .

Essas classes vão compilar? Vão rodar?

A resposta é sim. E, além de tudo, farão exatamente o que nós queremos, pois, quando `Gerente` e `Presidente` possuem os métodos perfeitamente implementados, a classe `Diretor` , que não possui o método implementado, vai usar a implementação herdada de `Gerente` .



E esse diagrama, no qual incluímos uma classe abstrata `Secretaria` sem o método `getBonificacao`, que é estendida por mais duas classes (`SecretariaAdministrativa`, `SecretariaAgencia`) que, por sua vez, implementam o método `getBonificacao`, vai compilar? Vai rodar?

De novo, a resposta é sim, pois `Secretaria` é uma classe abstrata e, por isso, o Java tem certeza de que ninguém vai conseguir instanciá-la e, muito menos, chamar o método `getBonificacao` dela. Lembrando que, nesse caso, não precisamos nem ao menos escrever o método abstrato `getBonificacao` na classe `Secretaria`.

Se eu não reescrever um método abstrato da minha classe mãe, o código não compilará. Mas posso, em vez disso, declarar a classe como abstrata!

JAVA.IO

Classes abstratas não possuem nenhum segredo no aprendizado, mas quem está aprendendo orientação a objetos pode ter uma enorme dificuldade para saber quando utilizá-las, o que é muito normal.

Estudaremos o pacote java.io, que usa bastantes classes abstratas, sendo um exemplo real de uso desse recurso, que vai melhorar o entendimento delas. (classe InputStream e suas filhas)

10.5 PARA SABER MAIS...

- Uma classe que estende uma classe normal também pode ser abstrata! Ela não poderá ser instanciada, mas sua classe pai sim!
- Uma classe abstrata não precisa necessariamente ter um método abstrato.

10.6 EXERCÍCIOS: CLASSES ABSTRATAS

1. Repare que a nossa classe `Conta` é uma excelente candidata para uma classe abstrata. Por quê? Que métodos seriam interessantes candidatos a serem abstratos?

   Transforme a classe `Conta` em abstrata:

   ```
   public abstract class Conta {
       // ...
   }
   ```

2. Como a classe `Conta` agora é abstrata, não conseguimos dar `new` nela mais. Se não podemos dar `new` em `Conta` , qual é a utilidade de ter um método que recebe uma referência a `Conta` como argumento? Aliás, posso ter isso?

3. Apenas para entender melhor o `abstract` , comente o método `getTipo()` da `ContaPoupanca` , dessa forma ele herdará o método diretamente de `Conta` .

   Transforme o método `getTipo()` da classe `Conta` em abstrato. Repare que, ao colocar a palavra chave `abstract` ao lado do método, o Eclipse rapidamente vai sugerir que você deve remover o corpo (body) do método com um quick fix.

   Sua classe `Conta` deve ficar parecida com:

```java
public abstract class Conta {
    // atributos e métodos que já existiam

    public abstract String getTipo();
}
```

Qual é o problema com a classe `ContaPoupanca` ?

4. Descomente o método `getTipo` na classe `ContaPoupanca` , e se necessário altere-o para que a classe possa compilar normalmente.

5. (opcional) Existe outra maneira de a classe `ContaPoupanca` compilar se você não reescrever o método abstrato?

6. (opcional) Pra que ter o método `getTipo` na classe `Conta` se ele não faz nada? O que acontece se simplesmente apagarmos esse método da classe `Conta` e deixarmos o método `getTipo` nas filhas?

7. (opcional) Posso chamar um método abstrato de dentro de um outro método da própria classe abstrata? Por exemplo, imagine que exista o seguinte método na classe `Conta` :

```java
public String recuperaDadosParaImpressao() {
   String dados = "Titular: " + this.titular;
   dados += "\nNúmero: " + this.numero;
   dados += "\nAgência: " + this.agencia;
   dados += "\nSaldo: R$" + this.saldo;
   return dados;
}
```

Podemos invocar o `getTipo` dentro deste método? Algo como:

```java
   dados += "\nTipo: " + this.getTipo();
```

CAPÍTULO 11

INTERFACES

"Uma imagem vale mil palavras. Uma interface vale mil imagens." -- Ben Shneiderman

Ao término desse capítulo, você será capaz de:

- dizer o que é uma interface e as diferenças entre herança e implementação;
- escrever uma interface em Java;
- utilizá-las como um poderoso recurso para diminuir acoplamento entre as classes.

11.1 AUMENTANDO NOSSO EXEMPLO

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos Gerentes, pelos Diretores do Banco. Então, teríamos uma classe `Diretor` :

```
public class Diretor extends Funcionario {

    public boolean autentica(int senha) {
        // verifica aqui se a senha confere com a recebida como parametro
    }

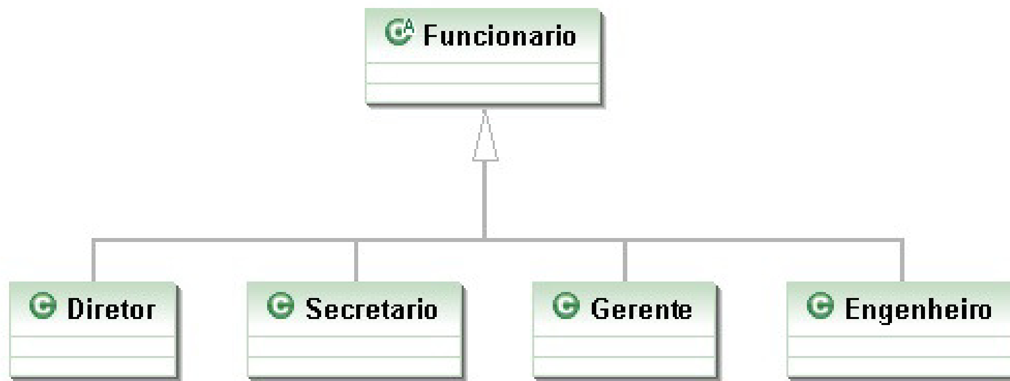
}
```

E a classe `Gerente` :

```
public class Gerente extends Funcionario {

    public boolean autentica(int senha) {
        // verifica aqui se a senha confere com a recebida como parametro
        // no caso do gerente verifica também se o departamento dele
        // tem acesso
    }

}
```



Repare que o método de autenticação de cada tipo de `Funcionario` pode variar muito. Mas vamos aos problemas. Considere o `SistemaInterno` e seu controle: precisamos receber um `Diretor` ou `Gerente` como argumento, verificar se ele se autentica e colocá-lo dentro do sistema.

```
public class SistemaInterno {

    public void login(Funcionario funcionario) {
        // invocar o método autentica?
        // não da! Nem todo Funcionario tem
    }
}
```

O `SistemaInterno` aceita qualquer tipo de `Funcionario`, tendo ele acesso ao sistema ou não, mas note que nem todo `Funcionario` possui o método `autentica`. Isso nos impede de chamar esse método com uma referência apenas a `Funcionario` (haveria um erro de compilação). O que fazer então?

```
public class SistemaInterno {

    public void login(Funcionario funcionario) {
        funcionario.autentica(...); // não compila
    }
}
```

Uma possibilidade é criar dois métodos `login` no `SistemaInterno`: um para receber `Diretor` e outro para receber `Gerente`. Já vimos que essa não é uma boa escolha. Por quê?

```
public class SistemaInterno {

    // design problemático
    public void login(Diretor funcionario) {
        funcionario.autentica(...);
    }

    // design problemático
    public void login(Gerente funcionario) {
        funcionario.autentica(...);
    }

}
```

Cada vez que criarmos uma nova classe de `Funcionario` que é *autenticável*, precisaríamos adicionar um novo método de login no `SistemaInterno` .

Métodos com mesmo nome

Em Java, métodos podem ter o mesmo nome desde que não sejam ambíguos, isto é, que exista uma maneira de distinguir no momento da chamada.

Isso se chama **sobrecarga** de método. (**Overloading**. Não confundir com **overriding**, que é um conceito muito mais poderoso).

Uma solução mais interessante seria criar uma classe no meio da árvore de herança, `FuncionarioAutenticavel` :

```
public class FuncionarioAutenticavel extends Funcionario {

    public boolean autentica(int senha) {
        // faz autenticacao padrão
    }

    // outros atributos e métodos

}
```

As classes `Diretor` e `Gerente` passariam a estender de `FuncionarioAutenticavel` , e o `SistemaInterno` receberia referências desse tipo, como a seguir:

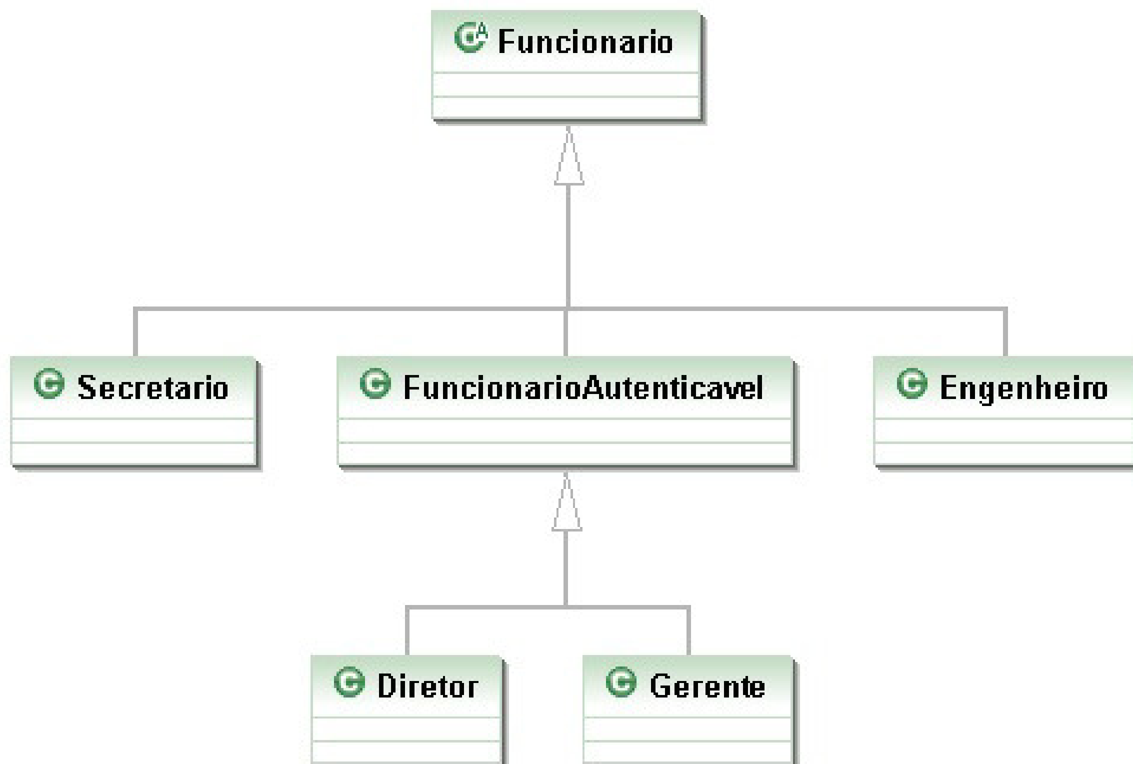
```
public class SistemaInterno {

    public void login(FuncionarioAutenticavel fa) {

        int senha = //pega senha de um lugar, ou de um scanner de polegar

        // aqui eu posso chamar o autentica!
        // Pois todo FuncionarioAutenticavel tem
        boolean ok = fa.autentica(senha);

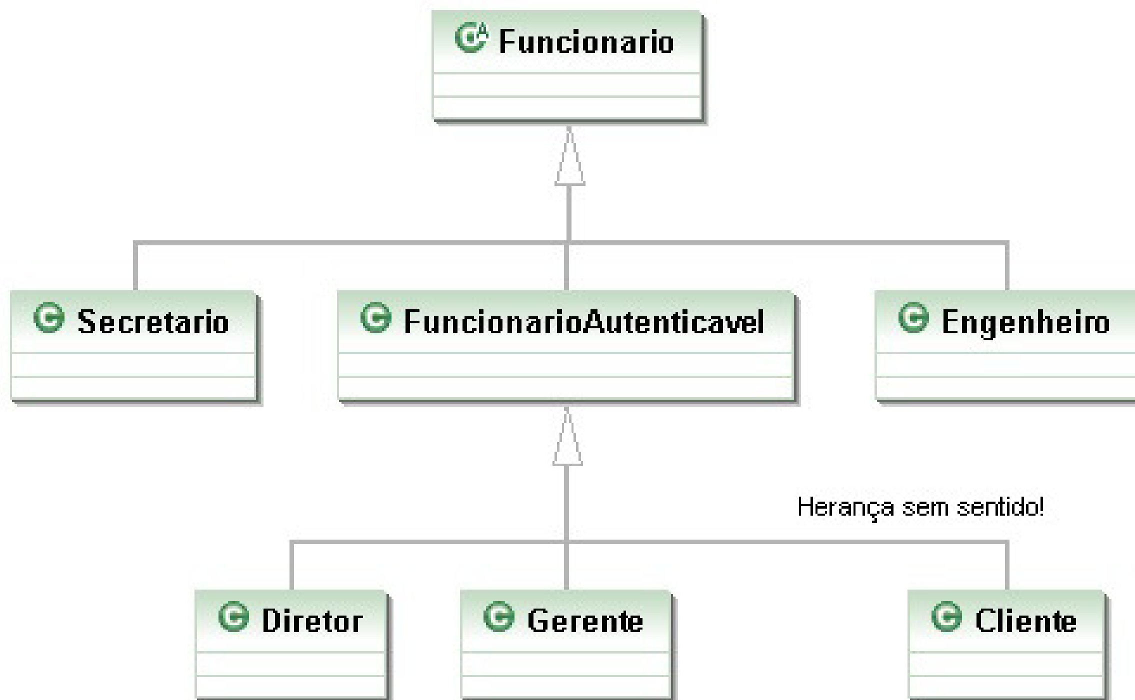
    }
}
```



Repare que `FuncionarioAutenticavel` é uma forte candidata a classe abstrata. Mais ainda, o método `autentica` poderia ser um método abstrato.

O uso de herança resolve esse caso, mas vamos a uma outra situação um pouco mais complexa: precisamos que todos os clientes também tenham acesso ao `SistemaInterno`. O que fazer? Uma opção é criar outro método `login` em `SistemaInterno`: mas já descartamos essa anteriormente.

Uma outra, que é comum entre os novatos, é fazer uma herança sem sentido para resolver o problema, por exemplo, fazer `Cliente extends FuncionarioAutenticavel`. Realmente, resolve o problema, mas trará diversos outros. `Cliente` definitivamente **não é** `FuncionarioAutenticavel`. Se você fizer isso, o `Cliente` terá, por exemplo, um método `getBonificacao`, um atributo salario e outros membros que não fazem o menor sentido para esta classe! Não faça herança quando a relação não é estritamente "é um".



Como resolver essa situação? Note que conhecer a sintaxe da linguagem não é o suficiente, precisamos estruturar/desenhar bem a nossa estrutura de classes.

11.2 INTERFACES

O que precisamos para resolver nosso problema? Arranjar uma forma de poder referenciar `Diretor` , `Gerente` e `Cliente` de uma mesma maneira, isto é, achar um fator comum.

Se existisse uma forma na qual essas classes garantissem a existência de um determinado método,

através de um contrato, resolveríamos o problema.

Toda classe define 2 itens:

- o que uma classe faz (as assinaturas dos métodos)
- como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)

Podemos criar um "contrato" que define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

```
contrato Autenticavel:

        quem quiser ser Autenticavel precisa saber fazer:
            1.autenticar dada uma senha, devolvendo um booleano
```

Quem quiser, pode "assinar" esse contrato, sendo assim obrigado a explicar como será feita essa autenticação. A vantagem é que, se um `Gerente` assinar esse contrato, podemos nos referenciar a um `Gerente` como um `Autenticavel`.

Podemos criar esse contrato em Java!

```
public interface Autenticavel {

    boolean autentica(int senha);

}
```

Chama-se `interface` pois é a maneira pela qual poderemos conversar com um `Autenticavel`. Interface é a maneira através da qual conversamos com um objeto.

Lemos a interface da seguinte maneira: *"quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano"*. Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Uma interface pode definir uma série de métodos, mas nunca conter implementação deles. Ela só expõe **o que o objeto deve fazer**, e não **como ele faz**, nem **o que ele tem**. **Como ele faz** vai ser definido em uma **implementação** dessa interface.

E o `Gerente` pode "assinar" o contrato, ou seja, **implementar** a interface. No momento em que ele implementa essa interface, ele precisa escrever os métodos pedidos pela interface (muito parecido com o efeito de herdar métodos abstratos, aliás, métodos de uma interface são públicos e abstratos, sempre). Para implementar usamos a palavra chave `implements` na classe:

```
public class Gerente extends Funcionario implements Autenticavel {

    private int senha;

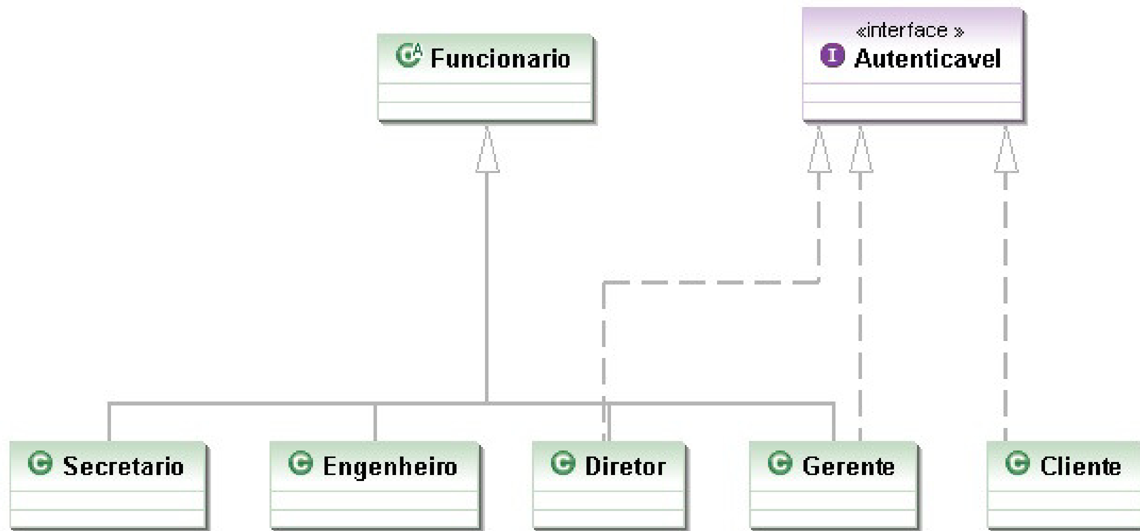
    // outros atributos e métodos

    public boolean autentica(int senha) {
```

```
        if(this.senha != senha) {
            return false;
        }
        // pode fazer outras possíveis verificações, como saber se esse
        // departamento do gerente tem acesso ao Sistema

        return true;
    }

}
```



O `implements` pode ser lido da seguinte maneira: "A classe `Gerente` se compromete a ser tratada como `Autenticavel`, sendo obrigada a ter os métodos necessários, definidos neste contrato".

A partir de agora, podemos tratar um `Gerente` como sendo um `Autenticavel`. Ganhamos mais polimorfismo! Temos mais uma forma de referenciar a um `Gerente`. Quando crio uma variável do tipo `Autenticavel`, estou criando uma referência para **qualquer** objeto de uma classe que implemente `Autenticavel`, direta ou indiretamente:

```
Autenticavel a = new Gerente();
// posso aqui chamar o método autentica!
```

Novamente, a utilização mais comum seria receber por argumento, como no nosso `SistemaInterno`:

```
public class SistemaInterno {

  public void login(Autenticavel a) {
     int senha = // pega senha de um lugar, ou de um scanner de polegar
     boolean ok =    a.autentica(senha);

     // aqui eu posso chamar o autentica!
     // não necessariamente é um Funcionario!
     // Mais ainda, eu não sei que objeto a
     // referência "a" está apontando exatamente! Flexibilidade.
  }
```

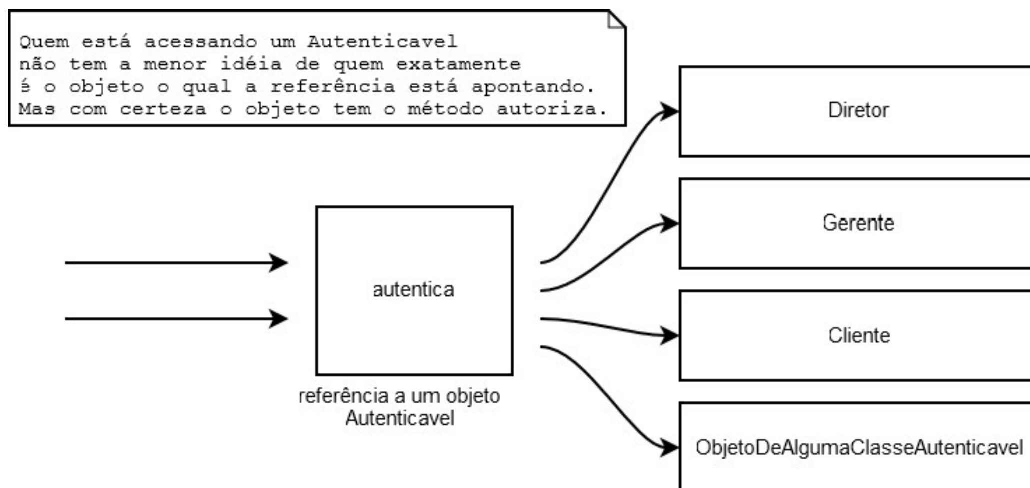
```
}
```

Pronto! E já podemos passar qualquer `Autenticavel` para o `SistemaInterno`. Então precisamos fazer com que o `Diretor` também implemente essa interface.

```
public class Diretor extends Funcionario implements Autenticavel {

    // métodos e atributos, além de obrigatoriamente ter o autentica

}
```



Podemos passar um `Diretor`. No dia em que tivermos mais um funcionário com acesso ao sistema, basta que ele implemente essa interface, para se encaixar no sistema.

Qualquer `Autenticavel` passado para o `SistemaInterno` está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método `autentica` que é o necessário para nosso `SistemaInterno` funcionar corretamente. Aliás, qualquer outra classe que futuramente implemente essa interface poderá ser passada como argumento aqui.

```
Autenticavel diretor = new Diretor();
Autenticavel gerente = new Gerente();
```

Ou, se achamos que o `Fornecedor` precisa ter acesso, basta que ele implemente `Autenticavel`. Olhe só o tamanho do desacoplamento: quem escreveu o `SistemaInterno` só precisa saber que ele é `Autenticavel`.

```
public class SistemaInterno {

  public void login(Autenticavel a) {
    // não importa se ele é um gerente ou diretor
    // será que é um fornecedor?
    // Eu, o programador do SistemaInterno, não me preocupo
    // Invocarei o método autentica
  }

}
```

Não faz diferença se é um `Diretor`, `Gerente`, `Cliente` ou qualquer classe que venha por aí. Basta seguir o contrato! Mais ainda, cada `Autenticavel` pode se autenticar de uma maneira completamente diferente de outro.

Lembre-se: a interface define que todos vão saber se autenticar (o que ele faz), enquanto a implementação define como exatamente vai ser feito (como ele faz).

A maneira como os objetos se comunicam num sistema orientado a objetos é muito mais importante do que como eles executam. **O que um objeto faz** é mais importante do que **como ele faz**. Aqueles que seguem essa regra, terão sistemas mais fáceis de manter e modificar. Como você já percebeu, esta é uma das ideias principais que queremos passar e, provavelmente, a mais importante de todo esse curso.

Mais sobre interfaces: herança e métodos default

Diferentemente das classes, uma interface pode herdar de mais de uma interface. É como um contrato que depende que outros contratos sejam fechados antes deste valer. Você não herda métodos e atributos, mas sim responsabilidades.

Um outro recurso em interfaces são os métodos default a partir do Java 8. Você pode sim declarar um método concreto, utilizando a palavra `default` ao lado, e suas implementações não precisam necessariamente reescrevê-lo. Veremos que isso acontece, por exemplo, com o método `List.sort`, durante o capítulo de coleções. É um truque muito utilizado para poder evoluir uma interface sem quebrar compatibilidade com as implementações anteriores.

11.3 DIFICULDADE NO APRENDIZADO DE INTERFACES

Interfaces representam uma barreira no aprendizado do Java: parece que estamos escrevendo um código que não serve pra nada, já que teremos essa linha (a assinatura do método) escrita nas nossas classes implementadoras. Essa é uma maneira errada de se pensar. O objetivo do uso de uma interface é deixar seu código mais flexível e possibilitar a mudança de implementação sem maiores traumas. **Não é apenas um código de prototipação, um cabeçalho**!

Os mais radicais dizem que toda classe deve ser "interfaceada", isto é, só devemos nos referir a objetos através de suas interfaces. Se determinada classe não tem uma interface, ela deveria ter. Os autores deste material acham tal medida radical demais, porém o uso de interfaces em vez de herança é amplamente aconselhado. Você pode encontrar mais informações sobre o assunto nos livros *Design Patterns*, *Refactoring* e *Effective Java*.

No livro Design Patterns, logo no início, os autores citam 2 regras "de ouro". Uma é "evite herança, prefira composição" e a outra, "programe voltado a interface e não à implementação".

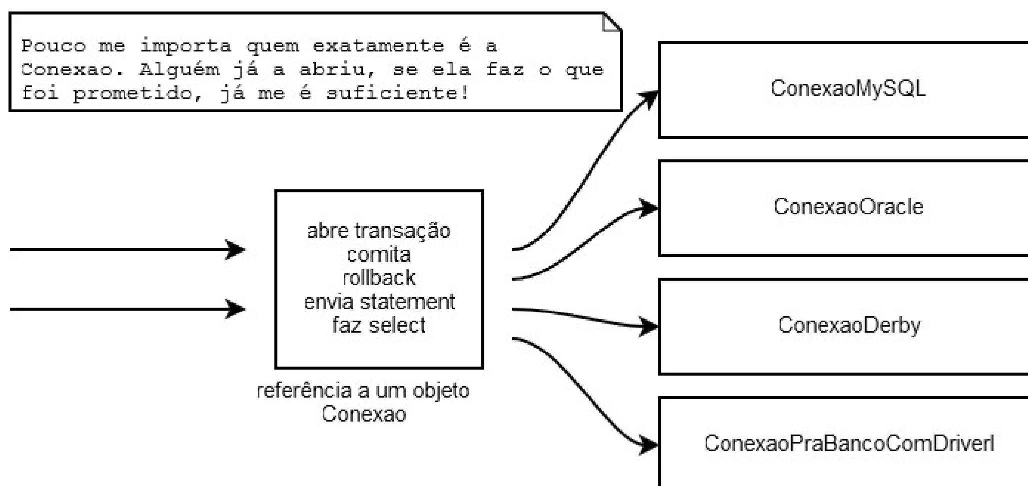
Veremos o uso de interfaces no capítulo de coleções, o que melhora o entendimento do assunto. O exemplo da interface `Comparable` também é muito esclarecedor, onde enxergamos o reaproveitamento de código através das interfaces, além do encapsulamento. Para o método `Collections.sort()` , pouco importa quem vai ser passado como argumento. Para ele, basta que a coleção seja de objetos comparáveis. Ele pode ordenar `Elefante` , `Conexao` ou `ContaCorrente` , desde que implementem `Comparable` .

11.4 EXEMPLO INTERESSANTE: CONEXÕES COM O BANCO DE DADOS

Como fazer com que todas as chamadas para bancos de dados diferentes respeitem a mesma regra? Usando interfaces!

Imagine uma interface `Conexao` contendo todos os métodos necessários para a comunicação e troca de dados com um banco de dados. Cada banco de dados fica encarregado de criar a sua implementação para essa interface.

Quem for usar uma `Conexao` não precisa se importar com qual objeto exatamente está trabalhando, já que ele vai cumprir o papel que toda `Conexao` deve ter. Não importa se é uma conexão com um Oracle ou MySQL.



Apesar do `java.sql.Connection` não trabalhar bem assim, a ideia é muito similar, porém as conexões vêm de uma *factory* chamada `DriverManager` .

Conexão a banco de dados está fora do escopo desse treinamento, mas é um dos primeiros tópicos abordados no curso FJ-21, juntamente com DAO.

Um pouco mais...

- Posso substituir toda minha herança por interfaces? Qual é a vantagem e a desvantagem?

11.5 EXERCÍCIOS: INTERFACES

1. Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso vamos criar uma interface no pacote `br.com.caelum.contas.modelo` do nosso projeto `fj11-contas` já existente:

   • o nome da interface deverá ser `Tributavel` ;

   • deve possuir um único método chamado `getValorImposto()` que não recebe nada e devolve um double.

   Lemos essa interface da seguinte maneira: "todos que quiserem ser *tributável* precisam saber retornar *o valor do imposto*, devolvendo um double".

   Alguns bens são tributáveis e outros não, `ContaPoupanca` não é tributável, já para `ContaCorrente` você precisa pagar 1% da conta e o `SeguroDeVida` tem uma taxa fixa de 42 reais mais 2% do valor do seguro.

   Assim, para atender esta nova necessidade, você deve:

   - *alterar* a classe `ContaCorrente` ;
   - *criar* a classe `SeguroDeVida` .

A classe `SeguroDeVida` deverá estar no pacote `br.com.caelum.contas.modelo` e deve ter os seguintes atributos encapsulados: `valor` (do tipo double), `titular` (do tipo String) e `numeroApolice` (do tipo int).

Dica: Na classe `SeguroDeVida`, lembre-se de escrever o método `getTipo` para que o tipo do `produto` apareça na interface gráfica.

2. Vamos criar a classe `ManipuladorDeSeguroDeVida` dentro do pacote `br.com.caelum.contas` para vincular a classe `SeguroDeVida` com a tela de criação de seguros. Esta classe deve ter um atributo do tipo `SeguroDeVida`.

   Deve ter também o método `criaSeguro` que não retorna nada e deve receber um parâmetro do tipo `Evento` para conseguir obter os dados da tela. Use os seguintes métodos da classe `Evento` para pegar estes dados:

   - `evento.getInt("numeroApolice"));`
   - `evento.getString("titular"));`
   - `evento.getDouble("valor"));`

   Dica: use os métodos "setters" da classe `SeguroDeVida` para guardar as informações obtidas. Exemplo:

```
this.seguroDeVida.setNumeroApolice(evento.getInt("numeroApolice"));
```

3. Execute a classe `TestaContas` e tente cadastrar um novo seguro de vida. O seguro cadastrado deve aparecer na tabela de seguros de vida.

4. Queremos saber qual o valor total dos impostos de todos os tributáveis. Vamos então criar a classe `ManipuladorDeTributaveis` dentro do pacote `br.com.caelum.contas`. Crie também o método `calculaImpostos` que não retorna nada e recebe um parâmetro do tipo `Evento`. Mais adiante preencheremos o corpo deste método.

   Esta classe também deverá ter o atributo encapsulado `total` do tipo double.

5. Agora que criamos o tributavel, vamos habilitar a última aba de nosso sistema. Altere a classe `TestaContas` para passar o valor `true` na chamada do método `mostraTela`.

   Observe que agora que temos o seguro de vida funcionando, a tela de relatório já consegue imprimir o valor dos impostos individuais de cada tipo de *Tributavel*.

6. No método `calculaImpostos` da classe `ManipuladorDeTributaveis` precisamos buscar os valores de impostos de cada `Tributavel`, somá-los e atribuir ao atributo `total`. Para isto, vamos usar os seguintes métodos da classe `Evento`:

- `getTamanhoDaLista` que deve receber o nome da lista desejada, no caso "listaTributaveis". Este método retorna a quantidade de tributáveis:

```
evento.getTamanhoDaLista("listaTributaveis");
```

- `getTributavel` retorna um `Tributavel` de uma determinada posição de uma lista, onde precisamos passar o nome da lista e o índice do elemento:

```
evento.getTributavel("listaTributaveis", i);
```

Dica: utilize o comando `for` para percorrer a lista inteira, passando por cada posição da mesma.

Por fim, o método `calculaImpostos` deverá invocar o método `getValorImposto()` e acumular o valor do imposto de todos os tributáveis no atributo `total` :

```
total += t.getValorImposto();
```

Repare que, de dentro do `ManipuladorDeTributaveis` , você não pode acessar o método `getSaldo` , por exemplo, pois você não tem a garantia de que o `Tributavel` que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface `Tributavel` .

É interessante enxergar que as interfaces (como aqui, no caso, `Tributavel`) costumam ligar classes muito distintas, unindo-as por uma característica que elas tem em comum. No nosso exemplo, `SeguroDeVida` e `ContaCorrente` são entidades completamente distintas, porém ambas possuem a característica de serem tributáveis.

Se amanhã o governo começar a tributar até mesmo `PlanoDeCapitalizacao` , basta que essa classe implemente a interface `Tributavel` ! Repare no grau de desacoplamento que temos: a classe `GerenciadorDeImpostoDeRenda` nem imagina que vai trabalhar como `PlanoDeCapitalizacao` . Para ela, o único fato que importa é que o objeto respeite o contrato de um tributável, isso é, a interface `Tributavel` . Novamente: programe voltado à interface, não à implementação.

Quais os benefícios de manter o código com baixo acoplamento?

7. (opcional) Crie a classe `TestaTributavel` com um método `main` para testar o nosso exemplo:

```
public class TestaTributavel {

    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        System.out.println(cc.getValorImposto());

        // testando polimorfismo:
        Tributavel t = cc;
        System.out.println(t.getValorImposto());
    }
```

```
}
```

Tente chamar o método `getSaldo` através da referência `t` , o que ocorre? Por quê?

A linha em que atribuímos `cc` a um `Tributavel` é apenas para você enxergar que é possível fazê-lo. Nesse nosso caso, isso não tem uma utilidade. Essa possibilidade foi útil no exercício anterior.

11.6 EXERCÍCIOS OPCIONAIS

Atenção: caso você faça esse exercício, faça isso num projeto à parte `conta-interface` já que usaremos a `Conta` como classe em exercícios futuros.

1. (Opcional) Transforme a classe `Conta` em uma interface.

   ```
   public interface Conta {
       public double getSaldo();
       public void deposita(double valor);
       public void saca(double valor);
       public void atualiza(double taxaSelic);
   }
   ```

   Adapte `ContaCorrente` e `ContaPoupanca` para essa modificação:

   ```
   public class ContaCorrente implements Conta {
       // ...
   }
   ```

   ```
   public class ContaPoupanca implements Conta {
       // ...
   }
   ```

   Algum código vai ter de ser copiado e colado? Isso é tão ruim?

2. (Opcional) Às vezes, é interessante criarmos uma interface que herda de outras interfaces: essas, são chamadas subinterfaces. Essas, nada mais são do que um agrupamento de obrigações para a classe que a implementar.

   ```
   public interface ContaTributavel extends Conta, Tributavel {
   }
   ```

   Dessa maneira, quem for implementar essa nova interface precisa implementar todos os métodos herdados das suas superinterfaces (e talvez ainda novos métodos declarados dentro dela):

   ```
   public class ContaCorrente implements ContaTributavel {
     // métodos
   }
   ```

   ```
   Conta c = new ContaCorrente();
   Tributavel t = new ContaCorrente();
   ```

   Repare que o código pode parecer estranho, pois a interface não declara método algum, só herda os métodos abstratos declarados nas outras interfaces.

Ao mesmo tempo que uma interface pode herdar de mais de uma outra interface, classes só podem possuir uma classe mãe (herança simples).

11.7 DISCUSSÃO: FAVOREÇA COMPOSIÇÃO EM RELAÇÃO À HERANÇA

Discuta com o instrutor e seus colegas, alternativas à herança. Falaremos de herança versus composição e porquê a herança é muitas vezes considerada maléfica.

Numa entrevista, James Gosling, "pai do java", fala sobre uma linguagem puramente de delegação e chega a dizer:

Rather than subclassing, just use pure interfaces. It's not so much that class inheritance is particularly bad. It just has problems.

(Tradução livre: *"Em vez de fazer subclasses, use simplesmente interfaces. Não é que a herança de classes seja particularmente ruim. Ela só tem problemas."*)

http://www.artima.com/intv/gosling3P.html

No blog da Caelum há também um post sobre o assunto: http://blog.caelum.com.br/2006/10/14/como-nao-aprender-orientacao-a-objetos-heranca/

CAPÍTULO 12

EXCEÇÕES E CONTROLE DE ERROS

"Quem pensa pouco, erra muito"--Leonardo da Vinci

Ao término desse capítulo, você será capaz de:

- controlar erros e tomar decisões baseadas nos mesmos;
- criar novos tipos de erros para melhorar o tratamento deles em sua aplicação ou biblioteca;
- assegurar que um método funcionou como diz em seu "contrato".

12.1 MOTIVAÇÃO

Voltando às `Conta` s que criamos no capítulo 6, o que aconteceria ao tentar chamar o método `saca` com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método `saca` não saberá que isso aconteceu.

Como avisar aquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Em Java, os métodos dizem qual o **contrato** que eles devem seguir. Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma `Conta` a ter um valor negativo, isto é, estar num estado inconsistente de acordo com a nossa modelagem.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
//     o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural do que a classe sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como `boolean` e retornar `true` , se tudo ocorreu da maneira planejada, ou `false` , caso contrário:

```
boolean saca(double quantidade) {
    // posso sacar até saldo+limite
    if (quantidade > this.saldo + this.limite) {
            System.out.println("Não posso sacar fora do limite!");
            return false;
        } else {
```

```
            this.saldo = this.saldo - quantidade;
            return true;
        }
    }
```

Um novo exemplo de chamada ao método acima:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
if (!minhaConta.saca(1000)) {
    System.out.println("Não saquei");
}
```

Repare que tivemos de lembrar de testar o retorno do método, mas não somos obrigados a fazer isso. Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de autoatendimento poderia vir a liberar a quantia desejada de dinheiro, mesmo que o sistema não tivesse conseguido efetuar o método `saca` com sucesso, como no exemplo a seguir:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);

// ...
double valor = 5000;
minhaConta.saca(valor); // vai retornar false, mas ninguém verifica!
caixaEletronico.emite(valor);
```

Mesmo invocando o método e tratando o retorno de maneira correta, o que faríamos se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução seria alterar o retorno de `boolean` para `int` e retornar o código do erro que ocorreu. Isso é considerado uma má prática (conhecida também como uso de "*magic numbers*").

Além de você perder o retorno do método, o valor devolvido é "mágico" e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando já num estado inconsistente.

Repare o que aconteceria se fosse necessário retornar um outro valor. O exemplo abaixo mostra um caso onde, através do retorno, não será possível descobrir se ocorreu um erro ou não, pois o método retorna um cliente.

```
public Cliente procuraCliente(int id) {
    if (idInvalido) {
        // avisa o método que chamou este que ocorreu um erro
    } else {
        Cliente cliente = new Cliente();
        cliente.setId(id);
        // cliente.setNome("nome do cliente");
        return cliente;
    }
}
```

Por esses e outros motivos, utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não iria acontecer. O exemplo do

argumento do saque inválido ou do `id` inválido de um cliente é uma **exceção** à regra.

Exceção

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

12.2 EXERCÍCIO PARA COMEÇAR COM OS CONCEITOS

Antes de resolvermos o nosso problema, vamos ver como a Java Virtual Machine age ao se deparar com situações inesperadas, como divisão por zero ou acesso a um índice da array que não existe.

1. Para aprendermos os conceitos básicos das exceptions do Java, teste o seguinte código você mesmo:

```
class TesteErro {
   public static void main(String[] args) {
       System.out.println("inicio do main");
       metodo1();
       System.out.println("fim do main");
   }

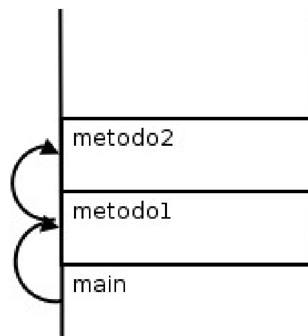
   static void metodo1() {
       System.out.println("inicio do metodo1");
       metodo2();
       System.out.println("fim do metodo1");
   }

   static void metodo2() {
       System.out.println("inicio do metodo2");
       ContaCorrente cc = new ContaCorrente();
       for (int i = 0; i <= 15; i++) {
           cc.deposita(i + 1000);
```

```
            System.out.println(cc.getSaldo());
            if (i == 5) {
                cc = null;
            }
        }
        System.out.println("fim do metodo2");
    }
}
```

Repare o método `main` chamando `metodo1` e esse, por sua vez, chamando o `metodo2`. Cada um desses métodos pode ter suas próprias variáveis locais, isto é: o `metodo1` não enxerga as variáveis declaradas dentro do `main` e por aí em diante.

Como o Java (e muitas das outras linguagens) faz isso? Toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da **pilha de execução** (*stack*): basta remover o marcador que está no topo da pilha:



Porém, o nosso `metodo2` propositadamente possui um enorme problema: está acessando uma referência nula quando o índice for igual a `6`!

Rode o código. Qual é a saída? O que isso representa? O que ela indica?

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
Exception in thread "main" java.lang.NullPointerException
        at br.com.caelum.contas.main.TesteErro.metodo2(TesteErro.java:22)
        at br.com.caelum.contas.main.TesteErro.metodo1(TesteErro.java:14)
        at br.com.caelum.contas.main.TesteErro.main(TesteErro.java:8)
```

Essa saída é o conhecido **rastro da pilha** (*stacktrace*). É uma saída importantíssima para o programador - tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa stacktrace. Mas por que isso aconteceu?

O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é **lançada** (*throw*), a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar** executar esse trecho de código. Como podemos ver, o `metodo2` não toma nenhuma medida

diferente do que vimos até agora.

Como o `metodo2` não está **tratando** esse problema, a JVM pára a execução dele anormalmente, sem esperar ele terminar, e volta um *stackframe* pra baixo, onde será feita nova verificação: "o `metodo1` está se precavendo de um problema chamado `NullPointerException` ?" "Não..." Volta para o `main` , onde também não há proteção, então a JVM morre (na verdade, quem morre é apenas a `Thread` corrente; se quiser saber mais sobre, há um apêndice de Threads e Programação Concorrente no final da apostila).

Obviamente, aqui estamos forçando esse caso e não faria sentido tomarmos cuidado com ele. É fácil arrumar um problema desses: basta verificar antes de chamar os métodos se a variável está com referência nula.

Porém, apenas para entender o controle de fluxo de uma `Exception` , vamos colocar o código que vai **tentar** (*try*) executar o bloco perigoso e, caso o problema seja do tipo `NullPointerException` , ele será **pego** (*caught*). Repare que é interessante que cada exceção no Java tenha um tipo... ela pode ter atributos e métodos.

2. Adicione um `try/catch` em volta do `for` , pegando `NullPointerException` . O que o código imprime?

```
try {
   for (int i = 0; i <= 15; i++) {
       cc.deposita(i + 1000);
       System.out.println(cc.getSaldo());
       if (i == 5) {
           cc = null;
       }
   }
} catch (NullPointerException e) {
   System.out.println("erro: " + e);
}
```

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
erro: java.lang.NullPointerException
fim do metodo2
fim do metodo1
fim do main
```

3. Em vez de fazer o `try` em torno do `for` inteiro, tente apenas com o bloco de dentro do `for` :

```
for (int i = 0; i <= 15; i++) {
    try {
        cc.deposita(i + 1000);
        System.out.println(cc.getSaldo());
        if (i == 5) {
            cc = null;
        }
    } catch (NullPointerException e) {
        System.out.println("erro: " + e);
    }
}
```

Qual é a diferença?

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
erro: java.lang.NullPointerException
fim do metodo2
fim do metodo1
fim do main
```

4. Retire o `try/catch` e coloque ele em volta da chamada do `metodo2` .

```
System.out.println("inicio do metodo1");
try {
    metodo2();
} catch (NullPointerException e) {
    System.out.println("erro: " + e);
}
System.out.println("fim do metodo1");
```

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
erro: java.lang.NullPointerException
fim do metodo1
fim do main
```

5. Faça o mesmo, retirando o `try/catch` novamente e colocando em volta da chamada do `metodo1`. Rode os códigos, o que acontece?

```java
System.out.println("inicio do main");
try {
    metodo1();
} catch (NullPointerException e) {
    System.out.println("erro: " + e);
}
System.out.println("fim do main");
```

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
erro: java.lang.NullPointerException
fim do main
```

Repare que, a partir do momento que uma exception foi *catched* (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

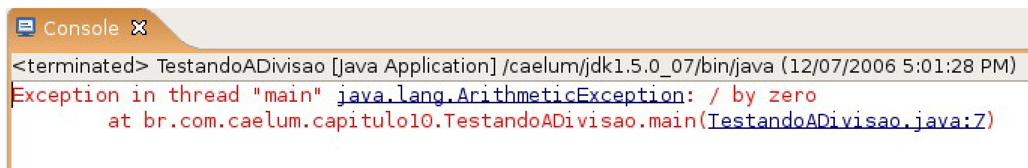
12.3 EXCEÇÕES DE RUNTIME MAIS COMUNS

Que tal tentar dividir um número por zero? Será que a JVM consegue fazer aquilo que nós definimos que não existe?

```
public class TestandoADivisao {

    public static void main(String[] args) {
        int i = 5571;
        i = i / 0;
        System.out.println("O resultado  " + i);
    }
}
```

Tente executar o programa acima. O que acontece?



Repare que um `NullPointerException` poderia ser facilmente evitado com um `if` que checaria se a referência é diferente de `null` .

Outro caso em que também ocorre tal tipo de exceção é quando um cast errado é feito (veremos mais pra frente). Em todos os casos, tais problemas provavelmente poderiam ser evitados pelo programador. É por esse motivo que o java não te obriga a dar o try/catch nessas exceptions e chamamos essas exceções de *unchecked*. Em outras palavras, o compilador não checa se você está tratando essas exceções.

ERROS

Os erros em Java são um tipo de exceção que também podem ser tratados. Eles representam problemas na máquina virtual e não devem ser tratados em 99% dos casos, já que provavelmente o melhor a se fazer é deixar a JVM encerrar (ou apenas a Thread em questão).

12.4 OUTRO TIPO DE EXCEÇÃO: CHECKED EXCEPTIONS

Fica claro, com os exemplos de código acima, que não é necessário declarar que você está tentando fazer algo onde um erro possa ocorrer. Os dois exemplos, com ou sem o `try/catch` , compilaram e rodaram. Em um, o erro terminou o programa e, no outro, foi possível tratá-lo.

Mas não é só esse tipo de exceção que existe em Java. Um outro tipo, obriga a quem chama o método ou construtor a tratar essa exceção. Chamamos esse tipo de exceção de *checked*, pois o compilador checará se ela está sendo devidamente tratada, diferente das anteriores, conhecidas como

unchecked.

Um exemplo interessante é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir (veremos como trabalhar com arquivos em outro capítulo, **não** se preocupe com isto agora):

```
class Teste {
    public static void metodo() {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

O código acima não compila e o compilador avisa que é necessário tratar o `FileNotFoundException` que pode ocorrer:

```
Teste.java:3: unreported exception java.io.FileNotFoundException; must be caught
 or declared to be thrown
                new java.io.FileReader("arquivo.txt");
                ^
1 error
```

Para compilar e fazer o programa funcionar, temos duas maneiras que podemos tratar o problema. O primeiro, é tratá-lo com o `try` e `catch` do mesmo jeito que usamos no exemplo anterior, de referência nula:

```
public static void metodo() {

    try {
        new java.io.FileInputStream("arquivo.txt");
    } catch (java.io.FileNotFoundException e) {
        System.out.println("Nao foi possível abrir o arquivo para leitura");
    }

}
```

A segunda forma de tratar esse erro, é delegar ele para quem chamou o nosso método, isto é, passar para a frente.

```
public static void metodo() throws java.io.FileNotFoundException {

    new java.io.FileInputStream("arquivo.txt");

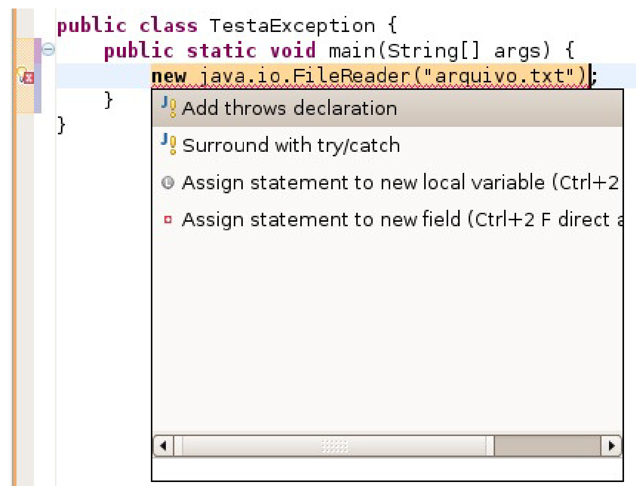
}
```

No Eclipse é bem simples fazer tanto um `try/catch` como um `throws` :

Tente digitar esse código no eclipse:

```
public class TestaException {
    public static void main(String[] args) {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

O Eclipse vai reclamar :



E você tem duas opções:

- *Add throws declaration*, que vai gerar:

```
public class TestaException {
    public static void main(String[] args) throws FileNotFoundException {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

- *Surround with try/catch*, que vai gerar:

```
public class TestaException2 {
    public static void main(String[] args) {
        try {
            new java.io.FileInputStream("arquivo.txt");
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

No início, existe uma grande tentação de sempre passar o problema pra frente para outros o tratarem. Pode ser que faça sentido, dependendo do caso, mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!

Não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

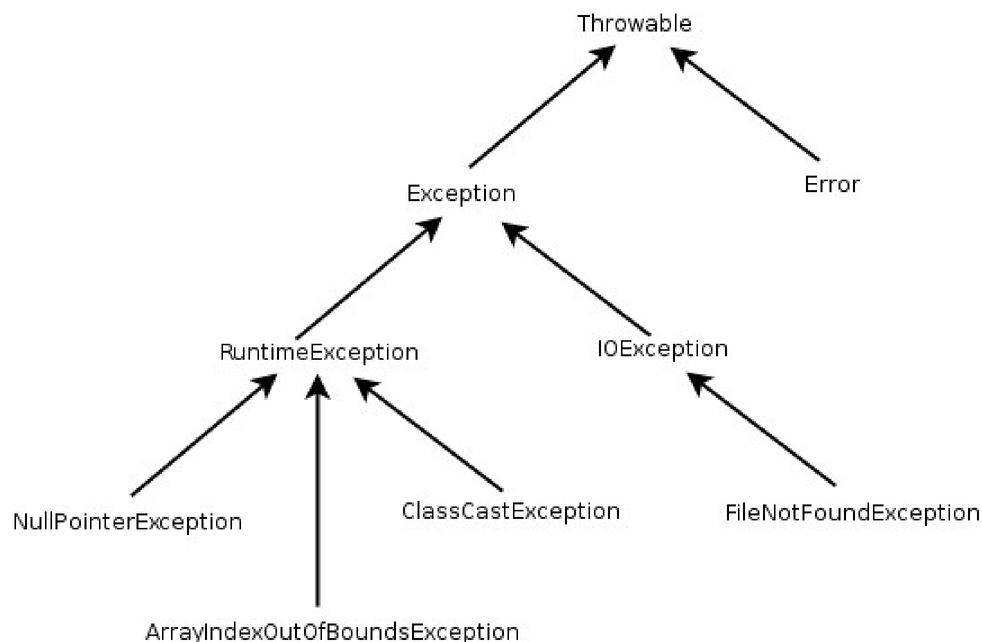
BOAS PRÁTICAS NO TRATAMENTO DE EXCEÇÕES

No blog da Caelum há um extenso artigo discutindo as boas práticas em relação ao tratamento de exceções.

http://blog.caelum.com.br/2006/10/07/lidando-com-exceptions/

12.5 UM POUCO DA GRANDE FAMÍLIA THROWABLE

Uma pequena parte da Família Throwable:



12.6 MAIS DE UM ERRO

É possível tratar mais de um erro quase que ao mesmo tempo:

- Com o try e catch:

```
try {
  objeto.metodoQuePodeLancarIOeSQLException();
} catch (IOException e) {
  // ..
} catch (SQLException e) {
  // ..
}
```

- Com o `throws` :

```
public void abre(String arquivo) throws IOException, SQLException {
  // ..
}
```

- Você pode, também, escolher tratar algumas exceções e declarar as outras no throws:

```
public void abre(String arquivo) throws IOException {
  try {
      objeto.metodoQuePodeLancarIOeSQLException();
  } catch (SQLException e) {
      // ..
  }
}
```

É desnecessário declarar no `throws` as exceptions que são *unchecked*, porém é permitido e às vezes, facilita a leitura e a documentação do seu código.

12.7 LANÇANDO EXCEÇÕES

Lembre-se do método `saca` da nossa classe `Conta` . Ele devolve um `boolean` caso consiga ou não sacar:

```
public boolean saca(double valor) {
    if (this.saldo < valor) {
        return false;
    } else {
        this.saldo-=valor;
        return true;
    }
}
```

Podemos, também, lançar uma `Exception` , o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um `if` no retorno de um método.

A palavra chave **throw**, que está no imperativo, lança uma `Exception` . Isto é bem diferente de `throws` , que está no presente do indicativo, e que apenas avisa da possibilidade daquele método lançá-la, obrigando o outro método que vá utilizar deste de se preocupar com essa exceção em questão.

```
public void saca(double valor) {
    if (this.saldo < valor) {
        throw new RuntimeException();
    } else {
        this.saldo-=valor;
    }
}
```

No nosso caso, lança uma do tipo *unchecked*. `RuntimeException` é a exception mãe de todas as exceptions *unchecked*. A desvantagem, aqui, é que ela é muito genérica; quem receber esse erro não sabe dizer exatamente qual foi o problema. Podemos então usar uma Exception mais específica:

```
public void saca(double valor) {
    if (this.saldo < valor) {
        throw new IllegalArgumentException();
    } else {
        this.saldo-=valor;
    }
}
```

`IllegalArgumentException` diz um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma Exception *unchecked* pois estende de `RuntimeException` e já faz parte da biblioteca do java. (`IllegalArgumentException` é a melhor escolha quando um argumento sempre é inválido como, por exemplo, números negativos, referências nulas, etc).

Para pegar esse erro, não usaremos um `if/else` e sim um `try/catch` , porque faz mais sentido já que a falta de saldo é uma exceção:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println("Saldo Insuficiente");
}
```

Podíamos melhorar ainda mais e passar para o construtor da `IllegalArgumentException` o motivo da exceção:

```
public void saca(double valor) {
    if (this.saldo < valor) {
        throw new IllegalArgumentException("Saldo insuficiente");
    } else {
        this.saldo-=valor;
    }
}
```

O método `getMessage()` definido na classe `Throwable` (mãe de todos os tipos de erros e exceptions) vai retornar a mensagem que passamos ao construtor da `IllegalArgumentException` .

```
try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
```

```
}
```


12.8 O QUE COLOCAR DENTRO DO TRY?

Imagine que vamos sacar dinheiro de diversas contas:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

Conta cp = new ContaPoupanca();
cp.deposita(100);

// sacando das contas:

cc.saca(50);
System.out.println("consegui sacar da corrente!");

cp.saca(50);
System.out.println("consegui sacar da poupança!");
```

Podemos escolher vários lugares para colocar try/catch:

```
try {
    cc.saca(50);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
System.out.println("consegui sacar da corrente!");

try {
    cp.saca(50);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
System.out.println("consegui sacar da poupança!");
```

Essa não parece uma opção boa, pois a mensagem *"consegui sacar"* será impressa mesmo que o `catch` seja acionado. Sempre que temos algo que depende da linha de cima para ser correto, devemos

agrupá-lo no `try` :

```
try {
    cc.saca(50);
    System.out.println("consegui sacar da corrente!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

try {
    cp.saca(50);
    System.out.println("consegui sacar da poupança!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

Mas há ainda uma outra opção: imagine que, para o nosso sistema, uma falha ao sacar da conta poupança deve parar o processo de saques e nem tentar sacar da conta corrente. Para isso, agruparíamos mais ainda:

```
try {
    cc.saca(50);
    System.out.println("consegui sacar da corrente!");
    cp.saca(50);
    System.out.println("consegui sacar da poupança!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

O que você vai colocar dentro do `try` influencia muito a execução do programa! Pense direito nas linhas que dependem uma da outra para a execução correta da sua lógica de negócios.

12.9 CRIANDO SEU PRÓPRIO TIPO DE EXCEÇÃO

É bem comum criar uma própria classe de exceção para controlar melhor o uso de suas exceções. Dessa maneira, podemos passar valores específicos para ela carregar, que sejam úteis de alguma forma. Vamos criar a nossa:

Voltamos para o exemplo das `Contas` , vamos criar a nossa Exceção de `SaldoInsuficienteException` :

```
public class SaldoInsuficienteException extends RuntimeException {

    public SaldoInsuficienteException(String message) {
        super(message);
    }
}
```

Em vez de lançar um `IllegalArgumentException` , vamos lançar nossa própria exception, com uma mensagem que dirá "Saldo Insuficiente":

```
public void saca(double valor) {
    if (this.saldo < valor) {
        throw new SaldoInsuficienteException("Saldo Insuficiente," +
```

```
                                            "tente um valor menor");
    } else {
        this.saldo-=valor;
    }
}
```

E, para testar, crie uma classe que deposite um valor e tente sacar um valor maior:

```
public static void main(String[] args) {
    Conta cc = new ContaCorrente();
    cc.deposita(10);

    try {
        cc.saca(100);
    } catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
    }
}
```

Podemos transformar essa `Exception` de *unchecked* para *checked*, obrigando a quem chama esse método a dar `try-catch`, ou `throws`:

```
public class SaldoInsuficienteException extends Exception {

    public SaldoInsuficienteException(String message) {
        super(message);
    }
}
```

12.10 PARA SABER MAIS: FINALLY

Os blocos `try` e `catch` podem conter uma terceira cláusula chamada `finally` que indica o que deve ser feito após o término do bloco `try` ou de um `catch` qualquer.

É interessante colocar algo que é imprescindível de ser executado, caso o que você queria fazer tenha dado certo, ou não. O caso mais comum é o de liberar um recurso no finally, como um arquivo ou conexão com banco de dados, para que possamos ter a certeza de que aquele arquivo (ou conexão) vá ser fechado, mesmo que algo tenha falhado no decorrer do código.

No exemplo a seguir, o bloco `finally` será sempre executado, independentemente de tudo ocorrer bem ou de acontecer algum problema:

```
try {
    // bloco try
} catch (IOException ex) {
    // bloco catch 1
} catch (SQLException sqlex) {
    // bloco catch 2
} finally {
    // bloco que será sempre executado, independente
    // se houve ou não exception e se ela foi tratada ou não
}
```

Há também, no Java 7, um recurso poderoso conhecido como *try-with-resources*, que permite

utilizar a semântica do finally de uma maneira bem mais simples.

12.11 EXERCÍCIOS: EXCEÇÕES

1. Na classe `Conta` , modifique o método `deposita(double x)` : Ele deve lançar uma exception chamada `IllegalArgumentException` , que já faz parte da biblioteca do Java, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

2. Rode a aplicação, cadastre uma conta e tente depositar um valor negativo. O que acontece?

3. Ao lançar a `IllegalArgumentException` , passe via construtor uma mensagem a ser exibida. Lembre que a `String` recebida como parâmetro é acessível depois via o método `getMessage()` herdado por todas as `Exceptions` .

   Rode a aplicação novamente e veja que agora a mensagem aparece na tela.

4. Faça o mesmo para o método `saca` da classe `ContaCorrente` , afinal o cliente também não pode sacar um valor negativo!

5. Vamos validar também que o cliente não pode sacar um valor maior do que o saldo disponível em conta. Crie sua própria `Exception` , `SaldoInsuficienteException` . Para isso, você precisa criar uma classe com esse nome que seja filha de `RuntimeException` .

   No método `saca` da classe `ContaCorrente` utilize esta nova `Exception` .

   Atenção: nem sempre é interessante criarmos um novo tipo de exception! Depende do caso. Neste aqui, seria melhor ainda utilizarmos `IllegalArgumentException` . A boa prática diz que devemos preferir usar as já existentes do Java sempre que possível.

6. (opcional) Coloque um construtor na classe `SaldoInsuficienteException` que receba o valor que ele tentou sacar (isto é, ele vai receber um `double valor`).

   Quando estendemos uma classe, não herdamos seus construtores, mas podemos acessá-los através da palavra chave `super` de dentro de um construtor. As exceções do Java possuem uma série de construtores úteis para poder populá-las já com uma mensagem de erro. Então vamos criar um construtor em `SaldoInsuficienteException` que delegue para o construtor de sua mãe. Essa vai guardar essa mensagem para poder mostrá-la ao ser invocado o método `getMessage` :

```
public class SaldoInsuficienteException extends RuntimeException {

    public SaldoInsuficienteException(double valor) {
        super("Saldo insuficiente para sacar o valor de: " + valor);
    }
}
```

   Dessa maneira, na hora de dar o `throw new SaldoInsuficienteException` você vai precisar passar esse valor como argumento:

```
if (this.saldo < valor) {
    throw new SaldoInsuficienteException(valor);
}
```

   Atenção: você pode se aproveitar do Eclipse para isso: comece já passando o `valor` como argumento para o construtor da exception e o Eclipse vai reclamar que não existe tal construtor. O quick fix (`ctrl + 1`) vai sugerir que ele seja construindo, poupando-lhe tempo!

   E agora, como fica o método `saca` da classe `ContaCorrente` ?

7. (opcional) Declare a classe `SaldoInsuficienteException` como filha direta de `Exception` em vez de `RuntimeException` . Ela passa a ser **checked**. O que isso resulta?

   Você vai precisar avisar que o seu método `saca()` `throws SaldoInsuficienteException` , pois ela é uma *checked* exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre `try-catch` ou `throws` . Faça uso do quick fix do Eclipse novamente!

   Depois, retorne a exception para *unchecked*, isto é, para ser filha de `RuntimeException` , pois utilizaremos ela assim em exercícios dos capítulos posteriores.

12.12 DESAFIOS

1. O que acontece se acabar a memória da java virtual machine?

12.13 DISCUSSÃO EM AULA: CATCH E THROWS EM EXCEPTION

Existe uma péssima prática de programação em java que é a de escrever o `catch` e o `throws` com

`Exception` .

Existem códigos que sempre usam `Exception` pois isso cuida de todos os possíveis erros. O maior problema disso é generalizar o erro. Se alguém joga algo do tipo `Exception` para quem o chamou, quem recebe não sabe qual o tipo específico de erro ocorreu e não vai saber como tratar o mesmo.

Sim, há casos onde o tratamento de mais de uma exception pode ser feito de uma mesma maneira. Por exemplo, se queremos terminar a aplicação tanto no caso de `IOException` quanto em `SQLException` . Se fizermos `catch(Exception e)` para pegar esses dois casos, teremos um problema: a aplicação vai parar mesmo que outra exceção seja lançada. A solução correta seria ter dois catches, mas aí teríamos código repetido. Para evitar o código repetido, podemos usar o multi-catch do Java 7, que permite um mesmo catch cuidar de mais de 1 exceção, através da sintaxe `catch(IOException | SQLException e) { ... }` .

CAPÍTULO 13

O PACOTE JAVA.LANG

"Nossas cabeças são redondas para que os pensamentos possam mudar de direção." -- Francis Piacaba

Ao término desse capítulo, você será capaz de:

- utilizar as principais classes do pacote `java.lang` e ler a documentação padrão de projetos java;
- usar a classe `System` para obter informações do sistema;
- utilizar a classe `String` de uma maneira eficiente e conhecer seus detalhes;
- utilizar os métodos herdados de `Object` para generalizar seu conceito de objetos.

13.1 PACOTE JAVA.LANG

Já usamos, por diversas vezes, as classes `String` e `System`. Vimos o sistema de pacotes do Java e nunca precisamos dar um `import` nessas classes. Isso ocorre porque elas estão dentro do pacote `java.lang`, que é **automaticamente importado** para você. É o **único pacote** com esta característica.

Vamos ver um pouco de suas principais classes.

13.2 UM POUCO SOBRE A CLASSE SYSTEM

A classe `System` possui uma série de atributos e métodos estáticos. Já usamos o atributo

`System.out` , para imprimir.

Olhando a documentação, você vai perceber que o atributo `out` é do tipo `PrintStream` do pacote `java.io` . Veremos sobre essa classe mais adiante. Já podemos perceber que poderíamos quebrar o `System.out.println` em duas linhas:

```
PrintStream saida = System.out;
saida.println("ola mundo!");
```

O `System` conta também com um método que simplesmente desliga a virtual machine, retornando um código de erro para o sistema operacional, é o `exit` .

```
    System.exit(0);
```

Veremos também um pouco mais sobre a classe `System` nos próximos capítulos e no apêndice de `Threads` . Consulte a documentação do Java e veja outros métodos úteis da `System` .

13.3 JAVA.LANG.OBJECT

Todo método que precisamos receber algum parâmetro temos que declarar o tipo do mesmo. Por exemplo, no nosso método `saca` precisamos passar como parâmetro um valor do tipo `double` . Se tentarmos passar qualquer coisa diferente disso teremos um erro de compilação.

Agora vamos observar o seguinte método do próprio Java:

```
System.out.println("Olá mundo!");
```

Neste caso, o método `println` está recebendo uma `String` e poderíamos pensar que o tipo de parâmetro que ele recebe é `String` . Mas ao mesmo tempo podemos passar para esse método coisas completamente diferentes como `int` , `Conta` , `Funcionario` , `SeguroDeVida` , etc. Como esse método consegue receber tantos parâmetros de tipos diferentes?

Uma possibilidade seria o uso da sobrecarga, declarando um `println` para cada tipo de objeto diferente. Mas claramente não é isso que acontece já que conseguimos criar uma classe qualquer e invocar o método `println` passando essa nova classe como parâmetro e ele funcionaria!

Para entender o que está acontecendo, vamos considerar um método que recebe uma `Conta` :

```
public void imprimeDados(Conta conta) {
    System.out.println(conta.getTitular() + " - " + conta.getSaldo());
}
```

Esse método pode ser invocado passando como parâmetro qualquer tipo de conta que temos no nosso sistema: `ContaCorrente` e `ContaPoupanca` pois ambas são filhas de `Conta` . Se quiséssemos que o nosso método conseguisse receber qualquer tipo de objeto teríamos que ter uma classe que fosse mãe de todos esses objetos. É para isso que existe a classe `Object` !

Sempre quando declaramos uma classe, essa classe é **obrigada** a herdar de outra. Isto é, para toda classe que declararmos, existe uma superclasse. Porém, criamos diversas classes sem herdar de ninguém:

```
class MinhaClasse {

}
```

Quando o Java não encontra a palavra chave `extends`, ele considera que você está herdando da classe `Object`, que também se encontra dentro do pacote `java.lang`. Você até mesmo pode escrever essa herança, que é o mesmo:

```
public class MinhaClasse extends Object {

}
```

Todas as classes, sem exceção, herdam de `Object`, seja direta ou indiretamente, pois ela é a mãe, vó, bisavó, etc de qualquer classe.

Podemos também afirmar que qualquer objeto em Java é um `Object`, podendo ser referenciado como tal. Então, qualquer objeto possui todos os métodos declarados na classe `Object` e veremos alguns deles logo após o *casting*.

13.4 MÉTODOS DO JAVA.LANG.OBJECT: EQUALS E TOSTRING

toString

A habilidade de poder se referir a qualquer objeto como `Object` nos traz muitas vantagens. Podemos criar um método que recebe um `Object` como argumento, isto é, qualquer objeto! Por exemplo, o método `println` poderia ser implementado da seguinte maneira:

```
public void println(Object obj) {
    write(obj.toString()); // o método write escreve uma string no console
}
```

Dessa forma, qualquer objeto que passarmos como parâmetro poderá ser impresso no console desde que ele possua o método `toString`. Para garantir que todos os objetos do Java possuam esse método, ele foi implementado na classe `Object`.

Por padrão, o método `toString` do `Object` retorna o nome da classe `@` um número de identidade:

```
Conta@34f5d74a
```

Mas e se quisermos imprimir algo diferente? Na nossa tela de detalhes de conta, temos uma caixa de seleção onde nossas contas estão sendo apresentadas com o valor do padrão do `toString`. Sempre que queremos modificar o comportamento de um método em relação a implementação herdada da

superclasse, podemos sobrescrevê-lo na classe filha:

```
public abstract class Conta {
    protected double saldo;
    // outros atributos...

    @Override
    public String toString() {
        return "[titular=" + titular + ", numero=" + numero
            + ", agencia=" + agencia + "]";
    }
}
```

Agora podemos usar esse método assim:

```
ContaCorrente cc = new ContaCorrente();
System.out.println(cc.toString());
```

E o melhor, se for apenas para jogar na tela, você nem precisa chamar o `toString` ! Ele já é chamado para você:

```
ContaCorrente cc = new ContaCorrente();
System.out.println(cc); // O toString é chamado pela classe PrintStream
```

Gera o mesmo resultado!

Você ainda pode concatenar `Strings` em Java com o operador `+` . Se o Java encontra um objeto no meio da concatenação, ele também chama o `toString` dele.

```
ContaCorrente cc = new ContaCorrente();
System.out.println("Conta: " + cc);
```

equals

Até agora estamos ignorando o fato que podemos mais de uma conta de mesmo número e agência no nosso sistema. Atualmente, quando inserimos uma nova conta, o sistema verifica se a conta inserida é igual a alguma outra conta já cadastrada. Mas qual critério de igualdade é utilizado por padrão para fazer essa verificação?

Assim como no caso do `toString` , todos objetos do Java possuem um outro método chamado `equals` que é utilizado para comparar objetos daquele tipo. Por padrão, esse método apenas compara as referências dos objetos. Como toda vez que inserimos uma nova conta no sistema estamos fazendo `new` em algum tipo de conta, as referências nunca vão ser iguais, mesmo os dados (número e agência) sendo iguais.

Mas, e se fosse preciso comparar os atributos? Quais atributos ele deveria comparar? O Java por si só não faz isso, mas podemos sobrescrever o `equals` da classe `Object` para criarmos esse critério de comparação.

O `equals` recebe um `Object` como argumento e deve verificar se ele mesmo é igual ao `Object`

recebido para retornar um `boolean` . Se você não reescrever esse método, o comportamento herdado é fazer um `==` com o objeto recebido como argumento.

```
public abstract class Conta {
    protected double saldo;
    // outros atributos...

    public boolean equals(Object object) {
        // primeiro verifica se o outro object não é nulo
        if (object == null) {
            return false;
        }

        if (this.numero == object.numero &&
            this.agencia.equals(object.agencia)) {
            return true;
        }
        return false;
    }
}
```

Casting de referências

No momento que recebemos uma referência para um `Object` , como vamos acessar os métodos e atributos desse objeto que imaginamos ser uma `Conta` ? Se estamos referenciando-o como `Object` , não podemos acessá-lo como sendo `Conta` . O código acima não compila!

Poderíamos então atribuir essa referência de `Object` para `Conta` para depois acessar os atributos necessários? Tentemos:

```
Conta outraConta = object;
```

Nós temos certeza de que esse `Object` se refere a uma `Conta` , já que a nossa lista só imprime contas. Mas o compilador Java não tem garantias sobre isso! Essa linha acima não compila, pois nem todo `Object` é uma `Conta` .

Para realizar essa atribuição, para isso devemos "avisar" o compilador Java que realmente queremos fazer isso, sabendo do risco que corremos. Fazemos o **casting de referências**, parecido com de tipos primitivos:

```
Conta outraConta = (Conta) object;
```

O código passa a compilar, mas será que roda? Esse código roda sem nenhum problema, pois em tempo de execução a JVM verificará se essa referência realmente é para um objeto de tipo `Conta` , e está! Se não estivesse, uma exceção do tipo `ClassCastException` seria lançada.

Com isso, nosso método `equals` ficaria assim:

```
public abstract class Conta {
    protected double saldo;
    // outros atributos...
```

```java
    public boolean equals(Object object) {
        if (object == null) {
            return false;
        }

        Conta outraConta = (Conta) object;
        if (this.numero == outraConta.numero &&
            this.agencia.equals(outraConta.agencia)) {
            return true;
        }
        return false;
    }
}
```

Você poderia criar um método com outro nome em vez de reescrever `equals` que recebe `Object` , mas ele é importante pois muitas bibliotecas o chamam através do polimorfismo, como veremos no capítulo do `java.util` .

O método `hashCode()` anda de mãos dadas com o método `equals()` e é de fundamental entendimento no caso de você utilizar suas classes com estruturas de dados que usam tabelas de espalhamento. Também falaremos dele no capítulo de `java.util` .

Regras para a reescrita do método equals

Pelo contrato definido pela classe `Object` devemos retornar `false` também no caso do objeto passado não ser de tipo compatível com a sua classe. Então antes de fazer o casting devemos verificar isso, e para tal usamos a palavra chave `instanceof` , ou teríamos uma exception sendo lançada.

Além disso, podemos resumir nosso `equals` de tal forma a não usar um `if` :

```java
public boolean equals(Object object) {
    if (object == null) {
        return false;
    }
    if (!(object instanceof Conta)) {
        return false;
    }
    Conta outraConta = (Conta) object;
    return (this.numero == outraConta.numero &&
        this.agencia.equals(outraConta.agencia));
}
```

13.5 EXERCÍCIOS: JAVA.LANG.OBJECT

1. Como verificar se a classe `Throwable` que é a superclasse de `Exception` também reescreve o método `toString` ?

   A maioria das classes do Java que são muito utilizadas terão seus métodos `equals` e `toString` reescritos convenientemente.

2. Utilize-se da documentação do Java e descubra de que classe é o objeto referenciado pelo atributo `out` da `System` .

   Repare que, com o devido `import` , poderíamos escrever:

   ```
   // falta a declaração da saída
   ________ saida = System.out;
   saida.println("ola");
   ```

   A variável `saida` precisa ser declarada de que tipo? É isso que você precisa descobrir. Se você digitar esse código no Eclipse, ele vai te sugerir um quickfix e declarará a variável para você.

   Estudaremos essa classe em um capítulo futuro.

3. Rode a aplicação e cadastre duas contas. Na tela de detalhes de conta, verifique o que aparece na caixa de seleção de conta para transferência. Por que isso acontece?

4. Reescreva o método `toString` da sua classe `Conta` fazendo com que uma mensagem mais explicativa seja devolvida. Lembre-se de aproveitar dos recursos do Eclipse para isto: digitando apenas o começo do nome do método a ser reescrito e pressionando **ctrl + espaço**, ele vai sugerir reescrever o método, poupando o trabalho de escrever a assinatura do método e cometer algum engano.

   ```
   public abstract class Conta {
   ```

```java
    protected double saldo;

      @Override
      public String toString() {
          return "[titular=" + titular + ", numero=" + numero
             + ", agencia=" + agencia + "]";
      }
    // restante da classe

  }
```

Rode a aplicação novamente, cadastre duas contas e verifique novamente a caixa de seleção da transferência. O que aconteceu?

5. Reescreva o método `equals` da classe `Conta` para que duas contas com o mesmo **número e agência** sejam consideradas iguais. Esboço:

```java
public abstract class Conta {

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }

        Conta outraConta = (Conta) obj;

        return this.numero == outraConta.numero &&
            this.agencia.equals(outraConta.agencia);
    }
}
```

Você pode usar o **ctrl + espaço** do Eclipse para escrever o esqueleto do método `equals`, basta digitar dentro da classe `equ` e pressionar **ctrl + espaço**.

Rode a aplicação e tente adicionar duas contas com o mesmo número e agência. O que acontece?

13.6 JAVA.LANG.STRING

`String` é uma classe em Java. Variáveis do tipo `String` guardam referências a objetos, e não um valor, como acontece com os tipos primitivos.

Aliás, podemos criar uma `String` utilizando o `new`:

```java
String x = new String("fj11");
String y = new String("fj11");
```

Criamos aqui, dois objetos diferentes. O que acontece quando comparamos essas duas referências utilizando o `==`?

```java
if (x == y) {
    System.out.println("referência para o mesmo objeto");
}
else {
    System.out.println("referências para objetos diferentes!");
```

```
}
```

Temos aqui dois objetos diferentes! E, então, como faríamos para verificar se o conteúdo do objeto é o mesmo? Utilizamos o método `equals`, que foi reescrito pela `String`, para fazer a comparação de `char` em `char`.

```
if (x.equals(y)) {
    System.out.println("consideramos iguais no critério de igualdade");
}
else {
    System.out.println("consideramos diferentes no critério de igualdade");
}
```

Aqui, a comparação retorna verdadeiro. Por quê? Pois quem implementou a classe `String` decidiu que este seria o melhor critério de comparação. Você pode descobrir os critérios de igualdade de cada classe pela documentação.

Podemos também concatenar `Strings` usando o `+`. Podemos concatenar `Strings` com qualquer objeto, até mesmo números:

```
int total = 5;
System.out.println("o total gasto é: " + total);
```

O compilador utilizará os métodos apropriados da classe `String` e possivelmente métodos de outras classes para realizar tal tarefa.

Se quisermos comparar duas Strings, utilizamos o método `compareTo`, que recebe uma `String` como argumento e devolve um inteiro indicando se a `String` vem antes, é igual ou vem depois da `String` recebida. Se forem iguais, é devolvido `0`; se for anterior à `String` do argumento, devolve um inteiro negativo; e, se for posterior, um inteiro positivo.

Fato importante: **uma String é imutável**. O java cria um pool de Strings para usar como cache e, se a `String` não fosse imutável, mudando o valor de uma `String` afetaria todas as `String` s de outras classes que tivessem o mesmo valor.

Repare no código abaixo:

```
String palavra = "fj11";
palavra.toUpperCase();
System.out.println(palavra);
```

Pode parecer estranho, mas ele imprime "fj11" em minúsculo. Todo método que parece alterar o valor de uma `String`, na verdade, cria uma nova `String` com as mudanças solicitadas e a retorna! Tanto que esse método não é `void`. O código realmente útil ficaria assim:

```
String palavra = "fj11";
String outra = palavra.toUpperCase();
System.out.println(outra);
```

Ou você pode eliminar a criação de outra variável temporária, se achar conveniente:

```
String palavra = "fj11";
palavra = palavra.toUpperCase();
System.out.println(palavra);
```

Isso funciona da mesma forma para **todos** os métodos que parecem alterar o conteúdo de uma `String` .

Se você ainda quiser trocar o número 1 para 2, faríamos:

```
String palavra = "fj11";
palavra = palavra.toUpperCase();
palavra = palavra.replace("1", "2");
System.out.println(palavra);
```

Ou ainda podemos concatenar as invocações de método, já que uma `String` é devolvida a cada invocação:

```
String palavra = "fj11";
palavra = palavra.toUpperCase().replace("1", "2");
System.out.println(palavra);
```

O funcionamento do pool interno de Strings do Java tem uma série de detalhes e você pode encontrar mais informações sobre isto na documentação da classe `String` e no seu método `intern()` .

Outros métodos da classe String

Existem diversos métodos da classe `String` que são extremamente importantes. Recomendamos sempre consultar o javadoc relativo a essa classe para aprender cada vez mais sobre a mesma.

Por exemplo, o método `charAt(i)` , retorna o caractere existente na posição `i` da String, o **método** `length` retorna o número de caracteres na mesma e o método `substring` que recebe um `int` e devolve a SubString a partir da posição passada por aquele `int` .

O `indexOf` recebe um char ou uma String e devolve o índice em que aparece pela primeira vez na String principal (há também o `lastIndexOf` que devolve o índice da última ocorrência).

O `toUpperCase` e o `toLowerCase` devolvem uma nova String toda em maiúscula e toda em minúscula, respectivamente.

A partir do Java 6, temos ainda o método `isEmpty` , que devolve `true` se a String for vazia ou `false` caso contrário.

Alguns métodos úteis para buscas são o `contains` e o `matches` .

Há muitos outros métodos, recomendamos que você sempre consulte o javadoc da classe.

JAVA.LANG STRINGBUFFER E STRINGBUILDER

Como a classe `String` é imutável, trabalhar com uma mesma `String` diversas vezes pode ter um efeito colateral: gerar inúmeras `String` s temporárias. Isto prejudica a performance da aplicação consideravelmente.

No caso de você trabalhar muito com a manipulação de uma mesma `String` (por exemplo, dentro de um laço), o ideal é utilizar a classe `StringBuffer` . A classe `StringBuffer` representa uma sequência de caracteres. Diferentemente da `String` , ela é mutável, e não possui aquele pool.

A classe `StringBuilder` tem exatamente os mesmos métodos, com a diferença dela não ser **thread-safe**. Veremos sobre este conceito no capítulo de Threads.String.

13.7 EXERCÍCIOS: JAVA.LANG.STRING

1. Queremos que as contas apresentadas na caixa de seleção da transferência apareçam com o nome do titular em maiúsculas. Para fazer isso vamos alterar o método `toString` da classe `Conta` . Utilize o método `toUpperCase` da `String` para isso.

2. Após alterarmos o método `toString` , aconteceu alguma mudança com o nome do titular que é apresentado na lista de contas? Por que?

3. Teste os exemplos desse capítulo, para ver que uma `String` é imutável. Por exemplo:

```
public class TestaString {

    public static void main(String[] args) {
        String s = "fj11";
        s.replaceAll("1", "2");
        System.out.println(s);
    }

}
```

   Como fazer para ele imprimir fj22?

4. Como fazer para saber se uma `String` se encontra dentro de outra? E para tirar os espaços em branco das pontas de uma `String` ? E para saber se uma `String` está vazia? E para saber quantos caracteres tem uma `String` ?

   Tome como hábito sempre pesquisar o JavaDoc! Conhecer a API, aos poucos, é fundamental para que você não precise reescrever a roda!

5. (opcional) Escreva um método que usa os métodos `charAt` e `length` de uma `String` para

imprimir a mesma caractere a caractere, com cada caractere em uma linha diferente.

6. (opcional) Reescreva o método do exercício anterior, mas modificando ele para que imprima a `String` de trás para a frente e em uma linha só. Teste-a para *"Socorram-me, subi no ônibus em Marrocos"* e *"anotaram a data da maratona"*.

7. (opcional) Pesquise a classe `StringBuilder` (ou StringBuffer no Java 1.4). Ela é mutável. Por que usá-la em vez da `String` ? Quando usá-la?

   Como você poderia reescrever o método de escrever a `String` de trás para a frente usando um `StringBuilder` ?

13.8 DESAFIO

1. Converta uma `String` para um número sem usar as bibliotecas do java que já fazem isso. Isso é, uma `String x = "762"` deve gerar um `int i = 762` .

   Para ajudar, saiba que um `char` pode ser "transformado" em `int` com o mesmo valor numérico fazendo:

```
char c = '3';
int i = c - '0'; // i vale 3!
```

   Aqui estamos nos aproveitando do conhecimento da tabela unicode: os números de 0 a 9 estão em sequência! Você poderia usar o método estático `Character.getNumericValue(char)` em vez disso.

13.9 DISCUSSÃO EM AULA: O QUE VOCÊ PRECISA FAZER EM JAVA?

Qual é a sua necessidade com o Java? Precisa fazer algoritmos de redes neurais? Gerar gráficos 3D? Relatórios em PDF? Gerar código de barra? Gerar boletos? Validar CPF? Mexer com um arquivo do Excel?

O instrutor vai mostrar que para a maioria absoluta das suas necessidades, alguém já fez uma biblioteca e a disponibilizou.

CAPÍTULO 14

UM POUCO DE ARRAYS

"O homem esquecerá antes a morte do pai que a perda da propriedade"--Maquiavel

Ao término desse capítulo, você será capaz de:

- declarar e instanciar arrays;
- popular e percorrer arrays.

14.1 O PROBLEMA

Dentro de um bloco, podemos declarar diversas variáveis e usá-las:

```
double saldoDaConta1 = conta1.getSaldo();
double saldoDaConta2 = conta2.getSaldo();
double saldoDaConta3 = conta3.getSaldo();
double saldoDaConta4 = conta4.getSaldo();
```

Isso pode se tornar um problema quando precisamos mudar a quantidade de variáveis a serem declaradas de acordo com um parâmetro. Esse parâmetro pode variar, como por exemplo, a quantidade de número contidos num bilhete de loteria. Um jogo simples possui 6 números, mas podemos comprar um bilhete mais caro, com 7 números ou mais.

Para facilitar esse tipo de caso podemos declarar um **vetor (array)** de doubles:

```
double[] saldosDasContas;
```

O `double[]` é um tipo. Uma array é sempre um objeto, portanto, a variável `saldosDasContas` é uma referência. Vamos precisar criar um objeto para poder usar a array. Como criamos o objeto-array?

```
saldosDasContas= new double[10];
```

O que fizemos foi criar uma array de double de 10 posições e atribuir o endereço no qual ela foi criada. Podemos ainda acessar as posições do array:

```
saldosDasContas[5] = conta5.getSaldo();
```

O código acima altera a sexta posição do array. No Java, os índices do array vão de 0 a n-1, onde n é o tamanho dado no momento em que você criou o array. Se você tentar acessar uma posição fora desse alcance, um erro ocorrerá durante a execução.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at ArrayIndexOutOfBoundsExceptionTeste.main(ArrayIndexOutOfBoundsExcepti
onTeste.java:5)
```

ARRAYS - UM PROBLEMA NO APRENDIZADO DE MUITAS LINGUAGENS

Aprender a usar arrays pode ser um problema em qualquer linguagem. Isso porque envolve uma série de conceitos, sintaxe e outros. No Java, muitas vezes utilizamos outros recursos em vez de arrays, em especial os pacotes de coleções do Java, que veremos no capítulo 15. Portanto, fique tranquilo caso não consiga digerir toda sintaxe das arrays num primeiro momento.

No caso do bilhete de loteria, podemos utilizar o mesmo recurso. Mais ainda, a quantidade de números do nosso bilhete pode ser definido por uma variável. Considere que `n` indica quantos números nosso bilhete terá, podemos então fazer:

```
int[] numerosDoBilhete = new int[n];
```

E assim podemos acessar e modificar os inteiros com índice de `0` a `n-1`.

14.2 ARRAYS DE REFERÊNCIAS

É comum ouvirmos "array de objetos". Porém quando criamos uma array de alguma classe, ela possui referências. O objeto, como sempre, está na memória principal e, na sua array, só ficam guardadas as **referências** (endereços).

```
ContaCorrente[] minhasContas;
minhasContas = new ContaCorrente[10];
```

Quantas contas foram criadas aqui? Na verdade, **nenhuma**. Foram criados 10 espaços que você pode utilizar para guardar uma referência a uma ContaCorrente. Por enquanto, eles se referenciam para lugar nenhum (`null`). Se você tentar:

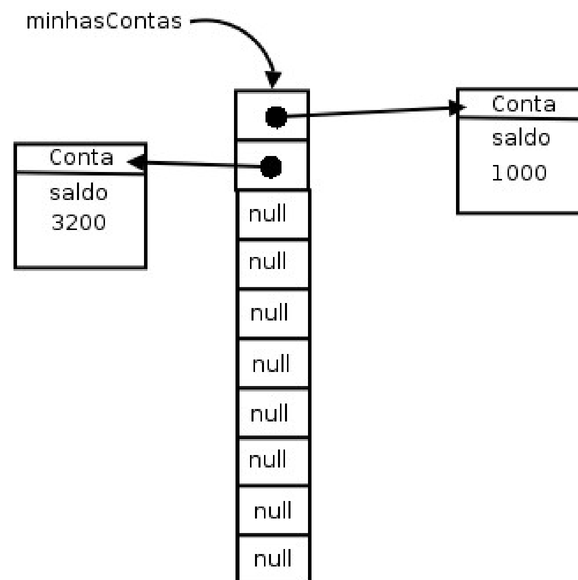
```
System.out.println(minhasContas[0].getSaldo());
```

Um erro durante a execução ocorrerá! Pois, na primeira posição da array, não há uma referência para uma conta, nem para lugar nenhum. Você deve **popular** sua array antes.

```
ContaCorrente contaNova = new ContaCorrente();
contaNova.deposita(1000.0);
minhasContas[0] = contaNova;
```

Ou você pode fazer isso diretamente:

```
minhasContas[1] = new ContaCorrente();
minhasContas[1].deposita(3200.0);
```



Uma array de tipos primitivos guarda valores, um array de objetos guarda referências.

Mas e se agora quisermos guardar tanto **Conta Corrente** quanto **Conta Poupança**? Um array de **Conta Corrente** só consegue guardar objetos do mesmo tipo. Se quisermos guardar os dois tipos de conta, devemos criar um array de **Conta**!

```
Conta[] minhasContas = new Conta[10];
minhasContas[0] = new ContaCorrente();
minhasContas[1] = new ContaPoupanca();
```

Perceba que não estamos criando um objeto do tipo `Conta` , que é abstrata, estamos criando 10 espaços que guardam referências para qualquer tipo de conta.

14.3 PERCORRENDO UMA ARRAY

Percorrer um array é muito simples quando fomos nós que a criamos:

```
public static void main(String[] args) {
    int[] idades = new int[10];
    for (int i = 0; i < 10; i++) {
        idades[i] = i * 10;
    }
    for (int i = 0; i < 10; i++) {
        System.out.println(idades[i]);
    }
}
```

Porém, em muitos casos, recebemos uma array como argumento em um método:

```
public void imprimeArray(int[] array) {
    // não compila!!
    for (int i = 0; i < ????; i++) {
        System.out.println(array[i]);
    }
}
```

Até onde o `for` deve ir? Toda array em Java tem um atributo que se chama `length`, e você pode acessá-lo para saber o tamanho do array ao qual você está se referenciando naquele momento:

```
public void imprimeArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}
```

ARRAYS NÃO PODEM MUDAR DE TAMANHO

A partir do momento que uma array foi criada, ela **não pode** mudar de tamanho.

Se você precisar de mais espaço, será necessário criar uma nova array e, antes de se referir ela, copie os elementos da array velha.

14.4 PERCORRENDO UMA ARRAY NO JAVA 5.0

O Java 5.0 traz uma nova sintaxe para percorremos arrays (e coleções, que veremos mais a frente).

No caso de você não ter necessidade de manter uma variável com o índice que indica a posição do elemento no vetor (que é uma grande parte dos casos), podemos usar o **enhanced-for**.

```
public class AlgumaClasse{
    public static void main(String[] args) {
        int[] idades = new int[10];
        for (int i = 0; i < 10; i++) {
            idades[i] = i * 10;
        }

        // imprimindo toda a array
```

```
        for (int x : idades) {
            System.out.println(x);
        }
    }
}
```

Não precisamos mais do `length` para percorrer matrizes cujo tamanho não conhecemos:

```
public class AlgumaClasse {
    public void imprimeArray(int[] array) {
        for (int x : array) {
            System.out.println(x);
        }
    }
}
```

O mesmo é válido para arrays de referências. Esse `for` nada mais é que um truque de compilação para facilitar essa tarefa de percorrer arrays e torná-la mais legível.

14.5 EXERCÍCIOS: ARRAYS

Para consolidarmos os conceitos sobre arrays, vamos fazer alguns exercícios que não interferem em nosso projeto.

1. Crie uma classe `TestaArrays` e no método `main` crie um array de contas de tamanho 10. Em seguida, faça um laço para criar 10 contas com saldos distintos e colocá-las no array. Por exemplo, você pode utilizar o índice do laço e multiplicá-lo por 100 para gerar o saldo de cada conta:

```
Conta[] contas = new Conta[10];

for (int i = 0; i < contas.length; i++) {
  Conta conta = new ContaCorrente();
  conta.deposita(i * 100.0);
  // escreva o código para guardar a conta na posição i do array
}
```

2. Ainda na classe `TestaArrays` , faça um outro laço para calcular e imprimir a média dos saldos de

todas as contas do array.

3. (opcional) Crie uma classe `TestaSplit` que reescreva uma frase com as palavras na ordem invertida. *"Socorram-me, subi no ônibus em Marrocos"* deve retornar *"Marrocos em ônibus no subi Socorram-me,"*. Utilize o método `split` da `String` para te auxiliar. Esse método divide uma `String` de acordo com o separador especificado e devolve as partes em um array de `String` , por exemplo:

```
String frase = "Uma mensagem qualquer";
String[] palavras = frase.split(" ");

// Agora só basta percorrer o array na ordem inversa imprimindo as palavras
```

4. (opcional) Crie uma classe `Banco` dentro do pacote `br.com.caelum.contas.modelo` O `Banco` deve ter um nome e um número (obrigatoriamente) e uma referência a uma array de `Conta` de tamanho 10, além de outros atributos que você julgar necessário.

```
public class Banco {
   private String nome;
   private int numero;
   private Conta[] contas;

   // outros atributos que você achar necessário

   public Banco(String nome, int numero) {
       this.nome = nome;
       this.numero = numero;
       this.contas = new ContaCorrente[10];
   }

   // getters para nome e número, não colocar os setters pois já recebemos no
   // construtor
}
```

5. (opcional) A classe `Banco` deve ter um método `adiciona` , que recebe uma referência a `Conta` como argumento e guarda essa conta.

   Você deve inserir a `Conta` em uma posição da array que esteja livre. Existem várias maneiras para você fazer isso: guardar um contador para indicar qual a próxima posição vazia ou procurar por uma posição vazia toda vez. O que seria mais interessante?

   Se quiser verificar qual a primeira posição vazia (nula) e adicionar nela, poderia ser feito algo como:

```
public void adiciona(Conta c) {
   for(int i = 0; i < this.contas.length; i++){
       // verificar se a posição está vazia
       // adicionar no array
   }
}
```

   É importante reparar que o método adiciona não recebe titular, agencia, saldo, etc. Essa seria uma maneira nem um pouco estruturada, muito menos orientada a objetos de se trabalhar. Você antes cria

uma `Conta` e já passa a referência dela, que dentro do objeto possui titular, saldo, etc.

6. (opcional) Crie uma classe `TestaBanco` que possuirá um método `main`. Dentro dele crie algumas instâncias de `Conta` e passe para o banco pelo método `adiciona`.

```
Banco banco = new Banco("CaelumBank", 999);
//     ....
```

Crie algumas contas e passe como argumento para o `adiciona` do banco:

```
ContaCorrente c1 = new ContaCorrente();
c1.setTitular("Batman");
c1.setNumero(1);
c1.setAgencia(1000);
c1.deposita(100000);
banco.adiciona(c1);

ContaPoupanca c2 = new ContaPoupanca();
c2.setTitular("Coringa");
c2.setNumero(2);
c2.setAgencia(1000);
c2.deposita(890000);
banco.adiciona(c2);
```

Você pode criar essas contas dentro de um loop e dar a cada um deles valores diferentes de depósitos:

```
for (int i = 0; i < 5; i++) {
   ContaCorrente conta = new ContaCorrente();
   conta.setTitular("Titular " + i);
   conta.setNumero(i);
   conta.setAgencia(1000);
   conta.deposita(i * 1000);
   banco.adiciona(conta);
}
```

Repare que temos de instanciar `ContaCorrente` dentro do laço. Se a instanciação de `ContaCorrente` ficasse acima do laço, estaríamos adicionado cinco vezes a **mesma** instância de `ContaCorrente` neste `Banco` e apenas mudando seu depósito a cada iteração, que nesse caso não é o efeito desejado.

Opcional: o método `adiciona` pode gerar uma mensagem de erro indicando quando o array já está cheio.

7. (opcional) Percorra o atributo `contas` da sua instância de `Banco` e imprima os dados de todas as suas contas. Para fazer isso, você pode criar um método chamado `mostraContas` dentro da classe `Banco`:

```
public void mostraContas() {
   for (int i = 0; i < this.contas.length; i++) {
       System.out.println("Conta na posição " + i);
       // preencher para mostrar outras informacoes da conta
   }
}
```

Cuidado ao preencher esse método: alguns índices do seu array podem não conter referência para uma `Conta` construída, isto é, ainda se referirem para `null`. Se preferir, use o `for` novo do java 5.0.

Aí, através do seu `main`, depois de adicionar algumas contas, basta fazer:

```
banco.mostraContas();
```

8. (opcional) Em vez de mostrar apenas o salário de cada funcionário, você pode usar o método `toString()` de cada `Conta` do seu array.

9. (opcional) Crie um método para verificar se uma determinada `Conta` se encontra ou não como conta deste banco:

```
public boolean contem(Conta conta) {
   // ...
}
```

Você vai precisar fazer um `for` em seu array e verificar se a conta passada como argumento se encontra dentro do array. Evite ao máximo usar números hard-coded, isto é, use o `.length`.

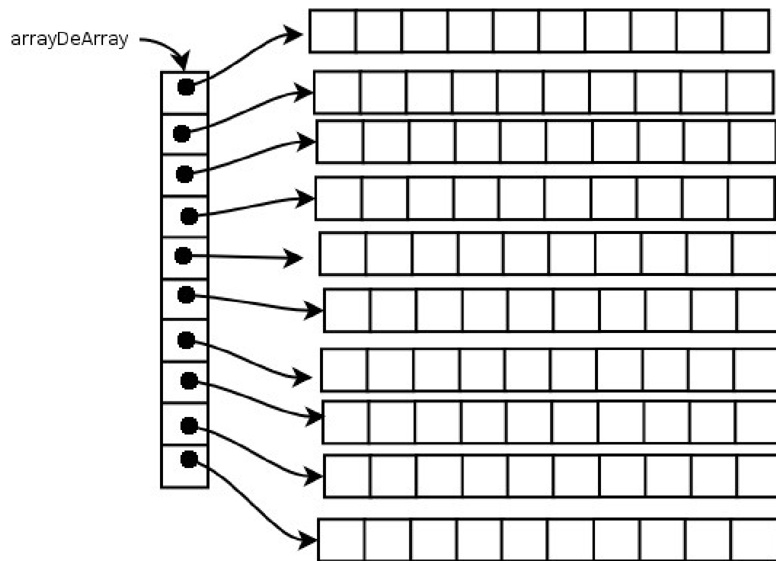
10. (opcional) Caso o array já esteja cheio no momento de adicionar uma outra conta, crie um array novo com uma capacidade maior e copie os valores do array atual. Isto é, vamos fazer a realocação dos elementos do array já que java não tem isso: um array nasce e morre com o mesmo length.

USANDO O THIS PARA PASSAR ARGUMENTO

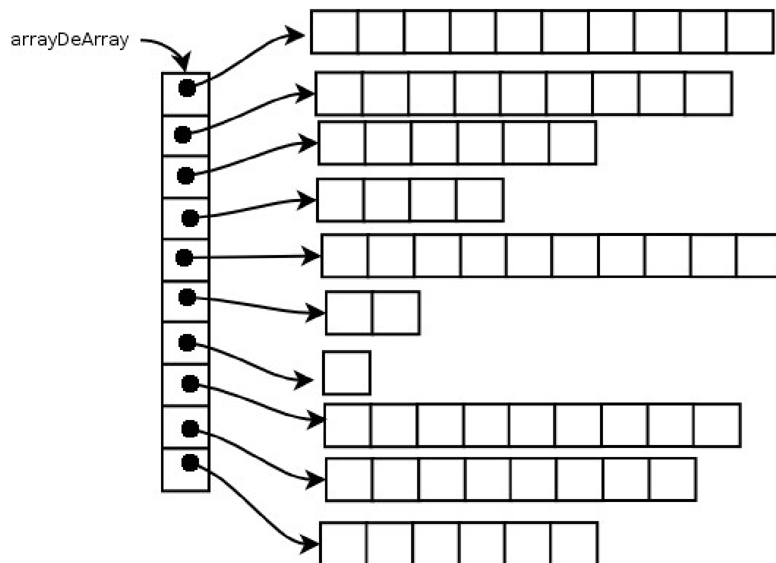
Dentro de um método, você pode usar a palavra `this` para referenciar a si mesmo e pode passar essa referência como argumento.

14.6 UM POUCO MAIS...

- Arrays podem ter mais de uma dimensão. Isto é, em vez de termos uma array de 10 contas, podemos ter uma array de 10 por 10 contas e você pode acessar a conta na posição da coluna x e linha y. Na verdade, uma array bidimensional em Java é uma array de arrays. Pesquise sobre isso.



- Uma array bidimensional não precisa ser retangular, isto é, cada linha pode ter um número diferente de colunas. Como? Porque?



- O que acontece se criarmos uma array de 0 elementos? e -1?

- O método `main` recebe uma **array de Strings** como argumento. Essa array é passada pelo usuário quando ele invoca o programa:

```
$ java Teste argumento1 outro maisoutro
```

E nossa classe:

```
public class Teste {
    public static void main (String[] args) {
        for(String argumento: args) {
```

```
            System.out.println(argumento);
        }
    }
}
```

Isso imprimirá:

```
argumento1
outro
maisoutro
```

14.7 DESAFIOS OPCIONAIS

1. Nos primeiros capítulos, você deve ter reparado que a versão recursiva para o problema de Fibonacci é lenta porque toda hora estamos recalculando valores. Faça com que a versão recursiva seja tão boa quanto a versão iterativa. (Dica: use arrays para isso)

2. O objetivo deste exercício é fixar os conceitos vistos. Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora no pequeno programa abaixo:

    - Programa:

      Classe: Casa Atributos: cor, totalDePortas, portas[] Métodos: void pinta(String s), int quantasPortasEstaoAbertas(), void adicionaPorta(Porta p), int totalDePortas()

      Crie uma casa, pinte-a. Crie três portas e coloque-as na casa através do método `adicionaPorta`, abra e feche-as como desejar. Utilize o método `quantasPortasEstaoAbertas` para imprimir o número de portas abertas e o método `totalDePortas` para imprimir o total de portas em sua casa.

CAPÍTULO 15

COLLECTIONS FRAMEWORK

"A amizade é um contrato segundo o qual nos comprometemos a prestar pequenos favores para que no-los retribuam com grandes." -- Baron de la Brede et de Montesquieu

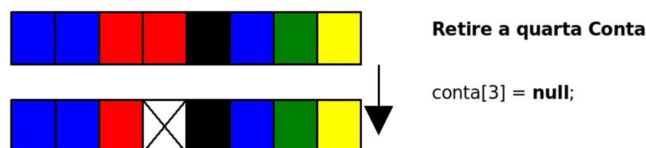
Ao término desse capítulo, você será capaz de:

- utilizar arrays, lists, sets ou maps dependendo da necessidade do programa;
- iterar e ordenar listas e coleções;
- usar mapas para inserção e busca de objetos.

15.1 ARRAYS SÃO TRABALHOSOS, UTILIZAR ESTRUTURA DE DADOS

Como vimos no capítulo de arrays, manipulá-las é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:

- não podemos redimensionar um array em Java;
- é impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
- não conseguimos saber quantas posições do array já foram populadas sem criar, para isso, métodos auxiliares.



Na figura acima, você pode ver um array que antes estava sendo completamente utilizado e que, depois, teve um de seus elementos removidos.

Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco? Precisaremos procurar por um espaço vazio? Guardaremos em alguma estrutura de dados externa, as posições vazias? E se não houver espaço vazio? Teríamos de criar um array maior e copiar os dados do antigo para ele?

Há mais questões: como posso saber quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array inteiro para conseguir essa informação?

Além dessas dificuldades que os arrays apresentavam, faltava um conjunto robusto de classes para suprir a necessidade de estruturas de dados básicas, como listas ligadas e tabelas de espalhamento.

Com esses e outros objetivos em mente, o comitê responsável pelo Java criou um conjunto de classes e interfaces conhecido como **Collections Framework**, que reside no pacote `java.util` desde o Java2 1.2.

COLLECTIONS

A **API** de **Collections** é robusta e possui diversas classes que representam estruturas de dados avançadas.

Por exemplo, não é necessário reinventar a roda e criar uma lista ligada, mas sim utilizar aquela que o Java disponibiliza.

15.2 LISTAS: JAVA.UTIL.LIST

Um primeiro recurso que a API de `Collections` traz são **listas**. Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.

Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos. Ela resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho "infinito",...). Esse código já está pronto!

A API de `Collections` traz a interface `java.util.List`, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis, cada uma com uma

forma diferente de representar uma lista.

A implementação mais utilizada da interface `List` é a `ArrayList`, que trabalha com um array interno para gerar uma lista. Portanto, ela é mais rápida na pesquisa do que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.

ARRAYLIST NÃO É UM ARRAY!

É comum confundirem uma `ArrayList` com um array, porém ela não é um array. O que ocorre é que, internamente, ela usa um array como estrutura para armazenar os dados, porém este atributo está propriamente encapsulado e você não tem como acessá-lo. Repare, também, que você não pode usar `[]` com uma `ArrayList`, nem acessar atributo `length`. Não há relação!

Para criar um `ArrayList`, basta chamar o construtor:

```
ArrayList lista = new ArrayList();
```

É sempre possível abstrair a lista a partir da interface `List`:

```
List lista = new ArrayList();
```

Para criar uma lista de nomes (`String`), podemos fazer:

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
```

A interface `List` possui dois métodos `add`, um que recebe o objeto a ser inserido e o coloca no final da lista, e um segundo que permite adicionar o elemento em qualquer posição da mesma. Note que, em momento algum, dizemos qual é o tamanho da lista; podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário.

Toda lista (na verdade, toda `Collection`) trabalha do modo mais genérico possível. Isto é, não há uma `ArrayList` específica para `Strings`, outra para Números, outra para Datas etc. Todos os métodos trabalham com `Object`.

Assim, é possível criar, por exemplo, uma lista de Contas Correntes:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(100);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(300);
```

```
List contas = new ArrayList();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

Para saber quantos elementos há na lista, usamos o método `size()` :

```
System.out.println(contas.size());
```

Há ainda um método `get(int)` que recebe como argumento o índice do elemento que se quer recuperar. Através dele, podemos fazer um `for` para iterar na lista de contas:

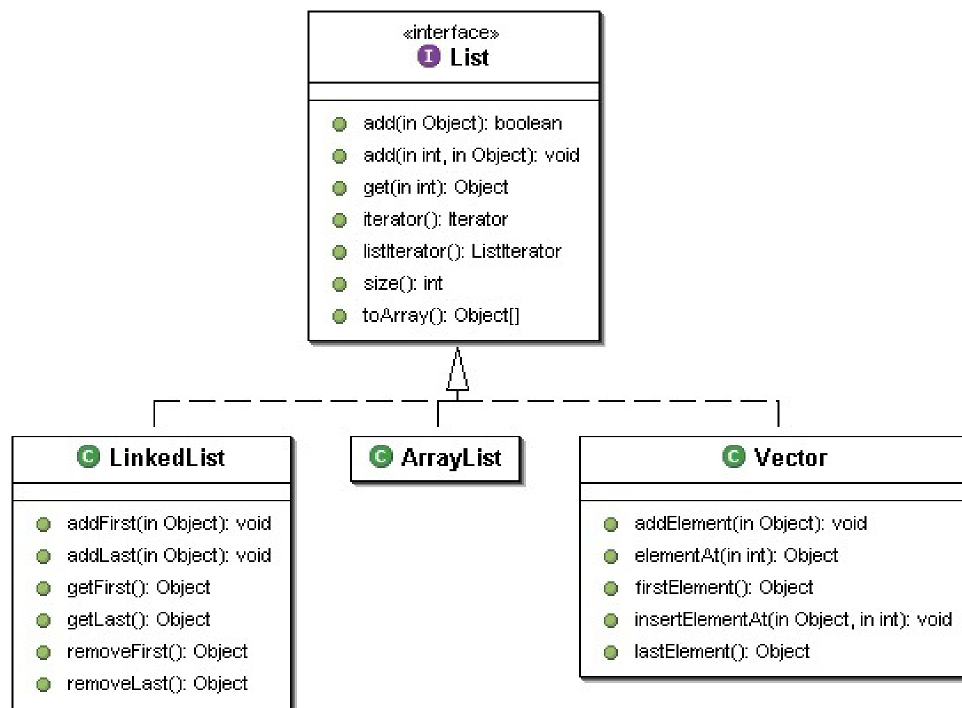
```
for (int i = 0; i < contas.size(); i++) {
    contas.get(i); // código não muito útil....
}
```

Mas como fazer para imprimir o saldo dessas contas? Podemos acessar o `getSaldo()` diretamente após fazer `contas.get(i)` ? Não podemos; lembre-se que toda lista trabalha sempre com `Object` . Assim, a referência devolvida pelo `get(i)` é do tipo `Object` , sendo necessário o cast para `ContaCorrente` se quisermos acessar o `getSaldo()` :

```
for (int i = 0; i < contas.size(); i++) {
    ContaCorrente cc = (ContaCorrente) contas.get(i);
    System.out.println(cc.getSaldo());
}
// note que a ordem dos elementos não é alterada
```

Há ainda outros métodos, como `remove()` que recebe um objeto que se deseja remover da lista; e `contains()` , que recebe um objeto como argumento e devolve `true` ou `false` , indicando se o elemento está ou não na lista.

A interface `List` e algumas classes que a implementam podem ser vistas no diagrama a seguir:



ACESSO ALEATÓRIO E PERCORRENDO LISTAS COM GET

Algumas listas, como a `ArrayList`, têm acesso aleatório aos seus elementos: a busca por um elemento em uma determinada posição é feita de maneira imediata, sem que a lista inteira seja percorrida (que chamamos de acesso sequencial).

Neste caso, o acesso através do método `get(int)` é muito rápido. Caso contrário, percorrer uma lista usando um `for` como esse que acabamos de ver, pode ser desastroso. Ao percorrermos uma lista, devemos usar **sempre** um `Iterator` ou `enhanced for`, como veremos.

Uma lista é uma excelente alternativa a um array comum, já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.

A outra implementação muito usada, a `LinkedList`, fornece métodos adicionais para obter e remover o primeiro e último elemento da lista. Ela também tem o funcionamento interno diferente, o que pode impactar performance, como veremos durante os exercícios no final do capítulo.

VECTOR

Outra implementação é a tradicional classe `Vector`, presente desde o Java 1.0, que foi adaptada para uso com o framework de Collections, com a inclusão de novos métodos.

Ela deve ser escolhida com cuidado, pois lida de uma maneira diferente com processos correndo em paralelo e terá um custo adicional em relação a `ArrayList` quando não houver acesso simultâneo aos dados.

15.3 LISTAS NO JAVA 5 E JAVA 7 COM GENERICS

Em qualquer lista, é possível colocar qualquer `Object`. Com isso, é possível misturar objetos:

```
ContaCorrente cc = new ContaCorrente();

List lista = new ArrayList();
lista.add("Uma string");
lista.add(cc);
...
```

Mas e depois, na hora de recuperar esses objetos? Como o método `get` devolve um `Object`, precisamos fazer o cast. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples...

Geralmente, não nos interessa uma lista com vários tipos de objetos misturados; no dia-a-dia, usamos listas como aquela de contas correntes. No Java 5.0, podemos usar o recurso de Generics para restringir as listas a um determinado tipo de objetos (e não qualquer `Object`):

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

Repare no uso de um parâmetro ao lado de `List` e `ArrayList`: ele indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo `ContaCorrente`. Isso nos traz uma segurança em tempo de compilação:

```
contas.add("uma string"); // isso não compila mais!!
```

O uso de Generics também elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo `ContaCorrente`:

```
for(int i = 0; i < contas.size(); i++) {
    ContaCorrente cc = contas.get(i); // sem casting!
    System.out.println(cc.getSaldo());
}
```

A partir do Java 7, se você instancia um tipo genérico na mesma linha de sua declaração, não é necessário passar os tipos novamente, basta usar `new ArrayList<>()`. É conhecido como *operador diamante*:

```
List<ContaCorrente> contas = new ArrayList<>();
```

15.4 A IMPORTÂNCIA DAS INTERFACES NAS COLEÇÕES

Vale ressaltar a importância do uso da interface `List`: quando desenvolvemos, procuramos sempre nos referir a ela, e não às implementações específicas. Por exemplo, se temos um método que vai buscar uma série de contas no banco de dados, poderíamos fazer assim:

```
class Agencia {
    public ArrayList<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<>();

        // para cada conta do banco de dados, contas.add

        return contas;
    }
}
```

Porém, para que precisamos retornar a referência específica a uma `ArrayList`? Para que ser tão específico? Dessa maneira, o dia que optarmos por devolver uma `LinkedList` em vez de `ArrayList`, as pessoas que estão usando o método `buscaTodasContas` poderão ter problemas, pois estavam fazendo referência a uma `ArrayList`. O ideal é sempre trabalhar com a interface mais genérica possível:

```
class Agencia {

    // modificação apenas no retorno:
    public List<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<>();

        // para cada conta do banco de dados, contas.add

        return contas;
    }
}
```

É o mesmo caso de preferir referenciar aos objetos com `InputStream` como fizemos no capítulo passado.

Assim como no retorno, é boa prática trabalhar com a interface em todos os lugares possíveis: métodos que precisam receber uma lista de objetos têm `List` como parâmetro em vez de uma implementação em específico como `ArrayList`, deixando o método mais flexível:

```
class Agencia {

    public void atualizaContas(List<Conta> contas) {
        // ...
```

```
    }
}
```

Também declaramos atributos como `List` em vez de nos comprometer como uma ou outra implementação. Dessa forma obtemos um **baixo acoplamento**: podemos trocar a implementação, já que estamos programando para a interface! Por exemplo:

```
class Empresa {

    private List<Funcionario> empregados = new ArrayList<>();

    // ...
}
```


15.5 ORDENAÇÃO: COLLECTIONS.SORT

Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas, muitas vezes, queremos percorrer a nossa lista de maneira ordenada.

A classe `Collections` traz um método estático `sort` que recebe um `List` como argumento e o ordena por ordem crescente. Por exemplo:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");

// repare que o toString de ArrayList foi sobrescrito:
System.out.println(lista);

Collections.sort(lista);

System.out.println(lista);
```

Ao testar o exemplo acima, você observará que, primeiro, a lista é impressa na ordem de inserção e, depois de invocar o `sort`, ela é impressa em ordem alfabética.

Mas toda lista em Java pode ser de qualquer tipo de objeto, por exemplo, `ContaCorrente`. E se quisermos ordenar uma lista de `ContaCorrente`? Em que ordem a classe `Collections` ordenará? Pelo saldo? Pelo nome do correntista?

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(500);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(150);

List<ContaCorrente> contas = new ArrayList<>();
contas.add(c1);
contas.add(c3);
contas.add(c2);

Collections.sort(contas); // qual seria o critério para esta ordenação?
```

Sempre que falamos em ordenação, precisamos pensar em um **critério de ordenação**, uma forma de determinar qual elemento vem antes de qual. É necessário instruir o `sort` sobre como **comparar** nossas `ContaCorrente` a fim de determinar uma ordem na lista. Para isto, o método `sort` necessita que todos seus objetos da lista sejam **comparáveis** e possuam um método que se compara com outra `ContaCorrente`. Como é que o método `sort` terá a garantia de que a sua classe possui esse método? Isso será feito, novamente, através de um contrato, de uma interface!

Vamos fazer com que os elementos da nossa coleção implementem a interface `java.lang.Comparable`, que define o método `int compareTo(Object)`. Este método deve retornar **zero**, se o objeto comparado for **igual** a este objeto, um número **negativo**, se este objeto for **menor** que o objeto dado, e um número **positivo**, se este objeto for **maior** que o objeto dado.

Para ordenar as `ContaCorrente`s por saldo, basta implementar o `Comparable`:

```
public class ContaCorrente extends Conta
                        implements Comparable<ContaCorrente> {

    // ... todo o código anterior fica aqui

    public int compareTo(ContaCorrente outra) {
      if (this.saldo < outra.saldo) {
        return -1;
      }

      if (this.saldo > outra.saldo) {
        return 1;
      }

      return 0;
    }
}
```

Com o código anterior, nossa classe tornou-se "**comparável**": dados dois objetos da classe,

conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido. No nosso caso, a comparação será feita baseando-se no saldo da conta.

Repare que o critério de ordenação é totalmente aberto, definido pelo programador. Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método `compareTo` na classe.

Quando chamarmos o método `sort` de `Collections`, ele saberá como fazer a ordenação da lista; ele usará o critério que definimos no método `compareTo`.

OUTRA IMPLEMENTAÇÃO...

O que acha da implementação abaixo?

```
public int compareTo(Conta outra) {
    return Integer.compare(this.getNumero(), outra.getNumero());
}
```

Mas, e o exemplo anterior, com uma lista de Strings? Por que a ordenação funcionou, naquele caso, sem precisarmos fazer nada? Simples: quem escreveu a classe `String` (lembre que ela é uma classe como qualquer outra) implementou a interface `Comparable` e o método `compareTo` para `String` s, fazendo comparação em ordem alfabética. (Consulte a documentação da classe `String` e veja o método `compareTo` lá). O mesmo acontece com outras classes como `Integer`, `BigDecimal`, `Date`, entre outras.

OUTROS MÉTODOS DA CLASSE COLLECTIONS

A classe `Collections` traz uma grande quantidade de métodos estáticos úteis na manipulação de coleções.

- `binarySearch(List, Object)` : Realiza uma busca binária por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado.
- `max(Collection)` : Retorna o maior elemento da coleção.
- `min(Collection)` : Retorna o menor elemento da coleção.
- `reverse(List)` : Inverte a lista.
- ...e muitos outros. Consulte a documentação para ver outros métodos.

No Java 8 muitas dessas funcionalidades da `Collections` podem ser feitas através dos chamados `Streams` , que fica um pouco fora do escopo de um curso inicial de Java.

Existe uma classe análoga, a `java.util.Arrays` , que faz operações similares com arrays.

É importante conhecê-las para evitar escrever código já existente.

15.6 EXERCÍCIOS: ORDENAÇÃO

Vamos ordenar o campo de **destino** da tela de detalhes da conta para que as contas apareçam em ordem alfabética de titular.

1. Faça sua classe `Conta` implementar a interface `Comparable<Conta>` . Utilize o critério de ordenar pelo titular da conta.

```
public class Conta implements Comparable<Conta> {
   ...
}
```

Deixe o seu método `compareTo` parecido com este:

```
public class Conta implements Comparable<Conta> {

   // ... todo o código anterior fica aqui

   public int compareTo(Conta outraConta) {
            return this.titular.compareTo(outraConta.titular);
   }
}
```

2. Queremos que as contas apareçam no campo de destino ordenadas pelo titular. Vamos então criar o método `ordenaLista` na classe `ManipuladorDeContas` . Use o `Collections.sort()` para ordenar a lista recuperada do `Evento` :

```
public class ManipuladorDeContas {

   // outros métodos

   public void ordenaLista(Evento evento) {
       List<Conta> contas = evento.getLista("destino");
       Collections.sort(contas);
   }
}
```

   Rode a aplicação, adicione algumas contas e verifique se as contas aparecem ordenadas pelo nome do titular, **no campo destino**, na parte da transferência. Para ver a ordenação é necessário acessar os detalhes de uma conta.

   Atenção especial: repare que escrevemos um método `compareTo` em nossa classe e nosso código **nunca** o invoca!! Isto é muito comum. Reescrevemos (ou implementamos) um método e quem o invocará será um outro conjunto de classes (nesse caso, quem está chamando o `compareTo` é o `Collections.sort` , que o usa como base para o algoritmo de ordenação). Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe `Collections` nunca imaginou que ordenaria objetos do tipo `Conta` , mas já que eles são `Comparable` , o seu método `sort` está satisfeito.

3. O que teria acontecido se a classe `Conta` não implementasse `Comparable<Conta>` mas tivesse o método `compareTo` ?

   Faça um teste: remova temporariamente a sentença `implements Comparable<Conta>` , não remova o método `compareTo` e veja o que acontece. Basta ter o método, sem assinar a interface?

4. Como posso inverter a ordem de uma lista? Como posso embaralhar todos os elementos de uma lista? Como posso rotacionar os elementos de uma lista?

   Investigue a documentação da classe `Collections` dentro do pacote `java.util` .

5. (opcional) Crie uma nova classe `TestaLista` que cria uma `ArrayList` e insere novas contas com saldos aleatórios usando um laço (`for`). Adivinhe o nome da classe para colocar saldos aleatórios? `Random` . Do pacote `java.util` . Consulte sua documentação para usá-la (utilize o método `nextInt()` passando o número máximo a ser sorteado).

6. Modifique a classe `TestaLista` para utilizar uma `LinkedList` em vez de `ArrayList` :

```
List<Conta> contas = new LinkedList<Conta>();
```

Precisamos alterar mais algum código para que essa substituição funcione? Rode o programa. Alguma diferença?

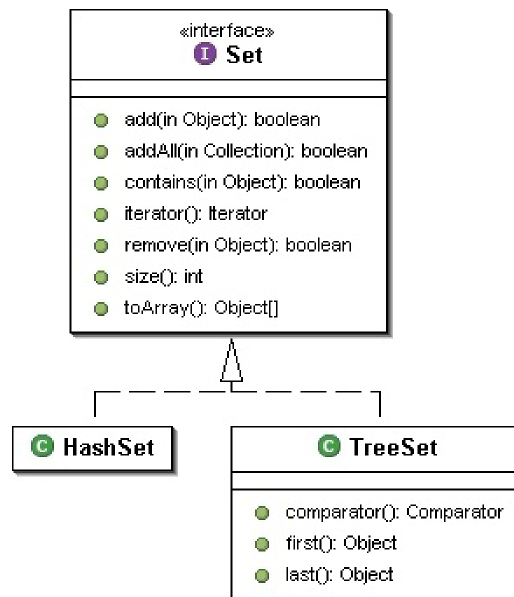
7. (opcional) Imprima a referência para essa lista. O `toString` de uma `ArrayList` / `LinkedList` é reescrito?

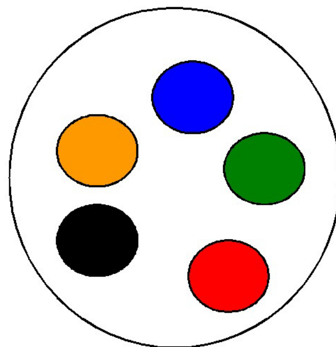
```
System.out.println(contas);
```

15.7 CONJUNTO: JAVA.UTIL.SET

Um conjunto (`Set`) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.

Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto. A interface não define como deve ser este comportamento. Tal ordem varia de implementação para implementação.





Possíveis ações em um conjunto:

- A camiseta Azul está no conjunto?
- Remova a camiseta Azul.
- Adicione a camiseta Vermelha.
- Limpe o conjunto.

- Não existem elementos duplicados!

- Ao percorrer um conjunto, sua ordem não é conhecida!

Um conjunto é representado pela interface `Set` e tem como suas principais implementações as classes `HashSet`, `LinkedHashSet` e `TreeSet`.

O código a seguir cria um conjunto e adiciona diversos elementos, e alguns repetidos:

```
Set<String> cargos = new HashSet<>();

cargos.add("Gerente");
cargos.add("Diretor");
cargos.add("Presidente");
cargos.add("Secretária");
cargos.add("Funcionário");
cargos.add("Diretor"); // repetido!

// imprime na tela todos os elementos
System.out.println(cargos);
```

Aqui, o segundo `Diretor` não será adicionado e o método `add` lhe retornará `false`.

O uso de um `Set` pode parecer desvantajoso, já que ele não armazena a ordem, e não aceita elementos repetidos. Não há métodos que trabalham com índices, como o `get(int)` que as listas possuem. A grande vantagem do `Set` é que existem implementações, como a `HashSet`, que possui uma performance incomparável com as `List` s quando usado para pesquisa (método `contains` por exemplo). Veremos essa enorme diferença durante os exercícios.

Ordem de um Set

Seria possível usar uma outra implementação de conjuntos, como um `TreeSet`, que insere os elementos de tal forma que, quando forem percorridos, eles apareçam em uma ordem definida pelo método de comparação entre seus elementos. Esse método é definido pela interface `java.lang.Comparable`. Ou, ainda, pode se passar um `Comparator` para seu construtor.

Já o `LinkedHashSet` mantém a ordem de inserção dos elementos.

Antes do Java 5, não podíamos utilizar generics, e usávamos o `Set` de forma que ele trabalhava com `Object`, havendo necessidade de castings.

15.8 PRINCIPAIS INTERFACES: JAVA.UTIL.COLLECTION

As coleções têm como base a interface `Collection` , que define métodos para adicionar e remover um elemento, e verificar se ele está na coleção, entre outras operações, como mostra a tabela a seguir:

`boolean add(Object)`	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna true ou false indicando se a adição foi efetuada com sucesso.
`boolean remove(Object)`	Remove determinado elemento da coleção. Se ele não existia, retorna `false`.
`int size()`	Retorna a quantidade de elementos existentes na coleção.
`boolean contains(Object)`	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método `equals()` do objeto, e não através do operador ==.
`Iterator iterator()`	Retorna um objeto que possibilita percorrer os elementos daquela coleção.

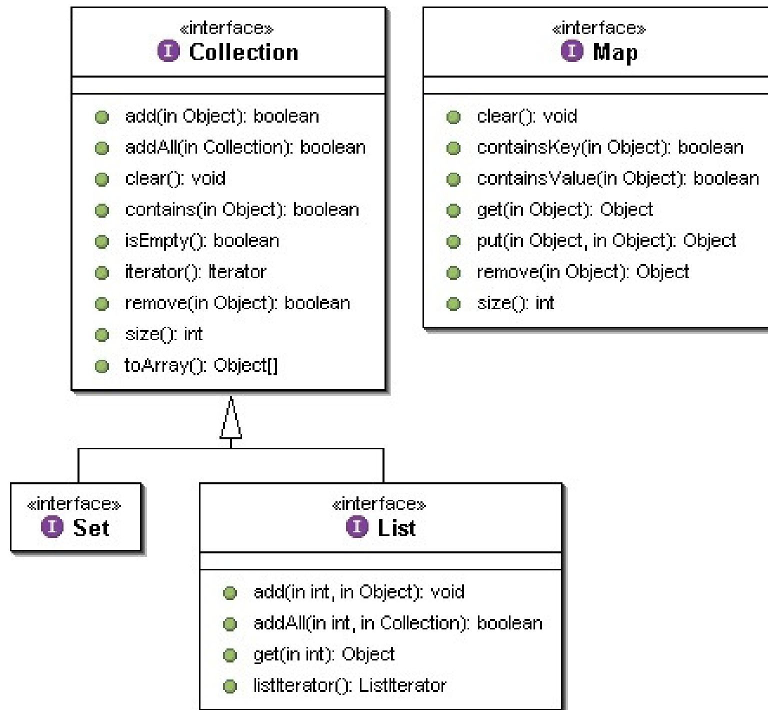
Uma coleção pode implementar diretamente a interface `Collection` , porém normalmente se usa uma das duas subinterfaces mais famosas: justamente `Set` e `List` .

A interface `Set` , como previamente vista, define um conjunto de elementos únicos enquanto a interface `List` permite elementos duplicados, além de manter a ordem a qual eles foram adicionados.

A busca em um `Set` pode ser mais rápida do que em um objeto do tipo `List` , pois diversas implementações utilizam-se de tabelas de espalhamento (*hash tables*), realizando a busca para tempo linear (**O(1)**).

A interface `Map` faz parte do framework, mas não estende `Collection` . (veremos `Map` mais

adiante).



No Java 5, temos outra interface filha de `Collection` : a `Queue` , que define métodos de entrada e de saída e cujo critério será definido pela sua implementação (por exemplo LIFO, FIFO ou ainda um heap onde cada elemento possui sua chave de prioridade).

15.9 PERCORRENDO COLEÇÕES NO JAVA 5

Como percorrer os elementos de uma coleção? Se for uma lista, podemos sempre utilizar um laço `for` , invocando o método `get` para cada elemento. Mas e se a coleção não permitir indexação?

Por exemplo, um `Set` não possui um método para pegar o primeiro, o segundo ou o quinto elemento do conjunto, já que um conjunto não possui o conceito de "ordem"

Podemos usar o **enhanced-for** (o "foreach") do Java 5 para percorrer qualquer `Collection` sem nos preocupar com isso. Internamente o compilador vai fazer com que seja usado o `Iterator` da `Collection` dada para percorrer a coleção. Repare:

```
Set<String> conjunto = new HashSet<>();

conjunto.add("java");
conjunto.add("vraptor");
conjunto.add("scala");

for (String palavra : conjunto) {
    System.out.println(palavra);
}
```

Em que ordem os elementos serão acessados?

Numa lista, os elementos aparecerão de acordo com o índice em que foram inseridos, isto é, de acordo com o que foi pré-determinado. Em um conjunto, a ordem depende da implementação da interface `Set` : você muitas vezes não vai saber ao certo em que ordem os objetos serão percorridos.

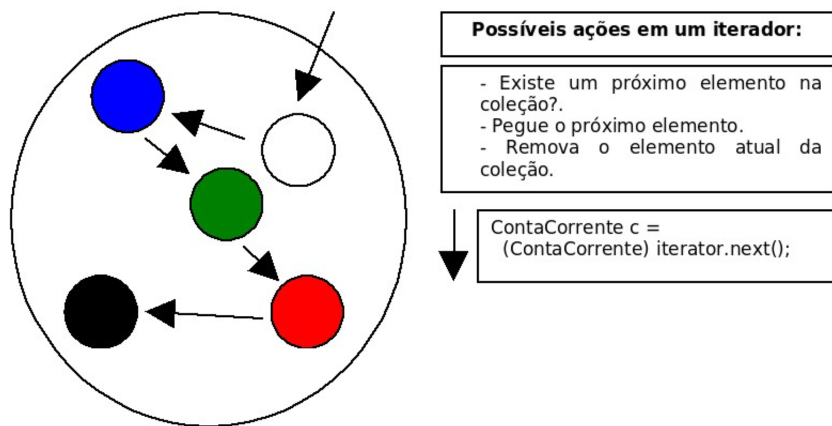
Por que o `Set` é, então, tão importante e usado?

Para perceber se um item já existe em uma lista, é muito mais rápido usar algumas implementações de `Set` do que um `List` , e os `TreeSets` já vêm ordenados de acordo com as características que desejarmos! Sempre considere usar um `Set` se não houver a necessidade de guardar os elementos em determinada ordem e buscá-los através de um índice.

No eclipse, você pode escrever `foreach` e dar **ctrl+espaço**, que ele vai gerar o esqueleto desse enhanced for! Muito útil!

15.10 PARA SABER MAIS: ITERANDO SOBRE COLEÇÕES COM JAVA.UTIL.ITERATOR

Antes do Java 5 introduzir o novo enhanced-for, iterações em coleções eram feitas com o `Iterator` . Toda coleção fornece acesso a um *iterator*, um objeto que implementa a interface `Iterator` , que conhece internamente a coleção e dá acesso a todos os seus elementos, como a figura abaixo mostra.



Ainda hoje (depois do Java 5) podemos usar o `Iterator` , mas o mais comum é usar o enhanced-for. E, na verdade, o enhanced-for é apenas um açúcar sintático que usa iterator por trás dos panos.

Primeiro criamos um `Iterator` que entra na coleção. A cada chamada do método `next` , o `Iterator` retorna o próximo objeto do conjunto. Um `iterator` pode ser obtido com o método `iterator()` de `Collection` , por exemplo numa lista de `String` :

```
Iterator<String> i = lista.iterator();
```

A interface `Iterator` possui dois métodos principais: `hasNext()` (com retorno booleano), indica se ainda existe um elemento a ser percorrido; `next()` , retorna o próximo objeto.

Voltando ao exemplo do conjunto de strings, vamos percorrer o conjunto:

```
Set<String> conjunto = new HashSet<>();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");

// retorna o iterator
Iterator<String> i = conjunto.iterator();
while (i.hasNext()) {
    // recebe a palavra
    String palavra = i.next();
    System.out.println(palavra);
}
```

O `while` anterior só termina quando todos os elementos do conjunto forem percorridos, isto é, quando o método `hasNext` mencionar que não existem mais itens.

ListIterator

Uma lista fornece, além de acesso a um `Iterator` , um `ListIterator` , que oferece recursos adicionais, específicos para listas.

Usando o `ListIterator` , você pode, por exemplo, adicionar um elemento na lista ou voltar para o elemento que foi "iterado" anteriormente.

Usar Iterator em vez do enhanced-for?

O `Iterator` pode sim ainda ser útil. Além de iterar na coleção como faz o enhanced-for, o `Iterator` consegue remover elementos da coleção durante a iteração de uma forma elegante, através do método `remove` .

Você pode também fazer o curso data dessa apostila na Caelum



Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

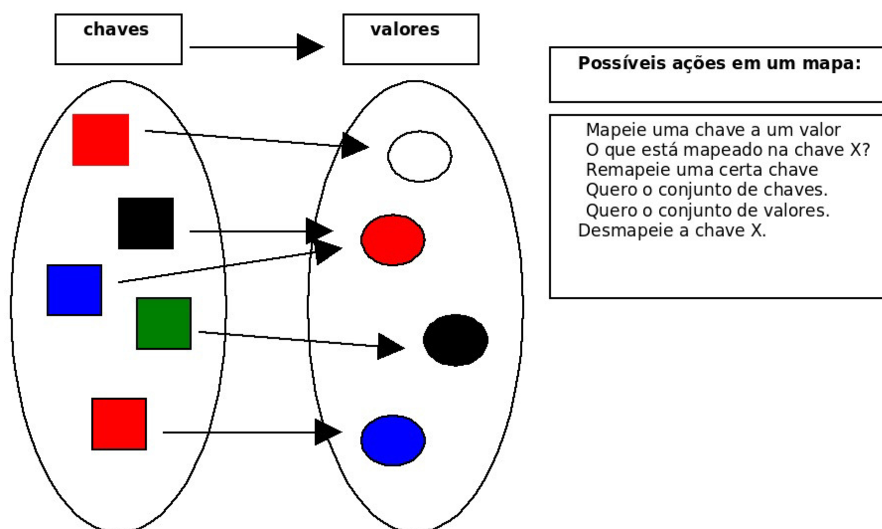
[Consulte as vantagens do curso *Java e Orientação a Objetos*]

15.11 MAPAS - JAVA.UTIL.MAP

Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa.

Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. É equivalente ao conceito de dicionário, usado em várias linguagens. Algumas linguagens, como Perl ou PHP, possuem um suporte mais direto a mapas, onde são conhecidos como matrizes/arrays associativas.

`java.util.Map` é um mapa, pois é possível usá-lo para mapear uma chave a um valor, por exemplo: mapeie à chave "empresa" o valor "Caelum", ou então mapeie à chave "rua" ao valor "Vergueiro". Semelhante a associações de palavras que podemos fazer em um dicionário.



O método `put(Object, Object)` da interface `Map` recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado objeto-chave, passa-se esse objeto no método `get(Object)`. Sem dúvida essas são as duas operações principais e mais frequentes realizadas sobre um mapa.

Observe o exemplo: criamos duas contas correntes e as colocamos em um mapa associando-as aos seus donos.

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);

// cria o mapa
Map<String, ContaCorrente> mapaDeContas = new HashMap<>();

// adiciona duas chaves e seus respectivos valores
mapaDeContas.put("diretor", c1);
mapaDeContas.put("gerente", c2);

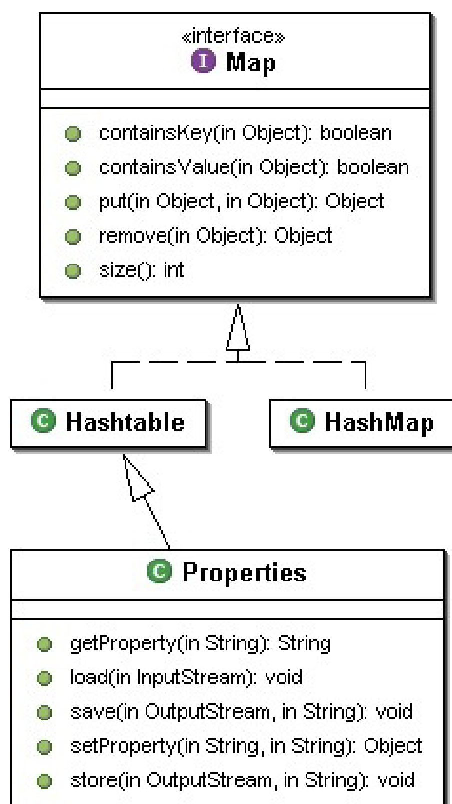
// qual a conta do diretor? (sem casting!)
ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

Um mapa é muito usado para "indexar" objetos de acordo com determinado critério, para podermos buscar esse objetos rapidamente. Um mapa costuma aparecer juntamente com outras coleções, para poder realizar essas buscas!

Ele, assim como as coleções, trabalha diretamente com `Objects` (tanto na chave quanto no valor), o que tornaria necessário o casting no momento que recuperar elementos. Usando os generics, como fizemos aqui, não precisamos mais do casting.

Suas principais implementações são o `HashMap`, o `TreeMap` e o `Hashtable`.

Apesar do mapa fazer parte do framework, ele não estende a interface `Collection`, por ter um comportamento bem diferente. Porém, as coleções internas de um mapa (a de chaves e a de valores, ver Figura 7) são acessíveis por métodos definidos na interface `Map`.



O método `keySet()` retorna um `Set` com as chaves daquele mapa e o método `values()` retorna a `Collection` com todos os valores que foram associados a alguma das chaves.

15.12 PARA SABER MAIS: PROPERTIES

Um mapa importante é a tradicional classe `Properties` , que mapeia strings e é muito utilizada para a configuração de aplicações.

A `Properties` possui, também, métodos para ler e gravar o mapeamento com base em um arquivo texto, facilitando muito a sua persistência.

```
Properties config = new Properties();

config.setProperty("database.login", "scott");
config.setProperty("database.password", "tiger");
config.setProperty("database.url","jdbc:mysql:/localhost/teste");

// muitas linhas depois...

String login = config.getProperty("database.login");
String password = config.getProperty("database.password");
String url = config.getProperty("database.url");
DriverManager.getConnection(url, login, password);
```

Repare que não houve a necessidade do casting para `String` no momento de recuperar os objetos associados. Isto porque a classe `Properties` foi desenhada com o propósito de trabalhar com a

associação entre `Strings` .

15.13 PARA SABER MAIS: EQUALS E HASHCODE

Muitas das coleções do java guardam os objetos dentro de tabelas de hash. Essas tabelas são utilizadas para que a pesquisa de um objeto seja feita de maneira rápida.

Como funciona? Cada objeto é "classificado" pelo seu `hashCode` e, com isso, conseguimos espalhar cada objeto agrupando-os pelo `hashCode` . Quando buscamos determinado objeto, só vamos procurar entre os elementos que estão no grupo daquele `hashCode` . Dentro desse grupo, vamos testando o objeto procurado com o candidato usando `equals()` .

Para que isso funcione direito, o método `hashCode` de cada objeto deve retornar o mesmo valor para dois objetos, se eles são considerados `equals` . Em outras palavras:

`a.equals(b)` implica `a.hashCode() == b.hashCode()`

Implementar `hashCode` de tal maneira que ele retorne valores diferentes para dois objetos considerados `equals` quebra o contrato de `Object` e resultará em collections que usam espalhamento (como `HashSet` , `HashMap e Hashtable`), não achando objetos iguais dentro de uma mesma coleção.

EQUALS E HASHCODE NO ECLIPSE

O Eclipse é capaz de gerar uma implementação correta de equals e hashcode baseado nos atributos que você queira comparar. Basta ir no menu Source e depois em Generate hashcode() and equals().

15.14 PARA SABER MAIS: BOAS PRÁTICAS

As coleções do Java oferecem grande flexibilidade ao usuário. A perda de performance em relação à utilização de arrays é irrelevante, mas deve-se tomar algumas precauções:

- Grande parte das coleções usam, internamente, um array para armazenar os seus dados. Quando esse array não é mais suficiente, é criada um maior e o conteúdo da antiga é copiado. Este processo pode acontecer muitas vezes, no caso de você ter uma coleção que cresce muito. Você deve, então, criar uma coleção já com uma capacidade grande, para evitar o excesso de redimensionamento.

- Evite usar coleções que guardam os elementos pela sua ordem de comparação quando não há necessidade. Um `TreeSet` gasta computacionalmente **`O(log(n))`** para inserir (ele utiliza uma árvore rubro-negra como implementação), enquanto o `HashSet` gasta apenas **`O(1)`** .

- Não itere sobre uma `List` utilizando um `for` de `0` até `list.size()` e usando `get(int)` para receber os objetos. Enquanto isso parece atraente, algumas implementações da `List` não são de acesso aleatório como a `LinkedList` , fazendo esse código ter uma péssima performance computacional. (use `Iterator`)

15.15 EXERCÍCIOS: COLLECTIONS

1. Crie um código que insira 30 mil números numa `ArrayList` e pesquise-os. Vamos usar um método de `System` para cronometrar o tempo gasto:

```
public class TestaPerformance {

    public static void main(String[] args) {
        System.out.println("Iniciando...");
        Collection<Integer> teste = new ArrayList<>();
```

```
        long inicio = System.currentTimeMillis();

        int total = 30000;

        for (int i = 0; i < total; i++) {
            teste.add(i);
        }

        for (int i = 0; i < total; i++) {
            teste.contains(i);
        }

        long fim = System.currentTimeMillis();
        long tempo = fim - inicio;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Troque a `ArrayList` por um `HashSet` e verifique o tempo que vai demorar:

```
Collection<Integer> teste = new HashSet<>();
```

O que é lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada `for` separadamente.

A diferença é mais que gritante. Se você passar de 30 mil para um número maior, como 50 ou 100 mil, verá que isso inviabiliza por completo o uso de uma `List`, no caso em que queremos utilizá-la essencialmente em pesquisas.

2. (conceitual, importante) Repare que, se você declarar a coleção e der `new` assim:

```
Collection<Integer> teste = new ArrayList<>();
```

em vez de:

```
ArrayList<Integer> teste = new ArrayList<>();
```

É garantido que vai ter de alterar só essa linha para substituir a implementação por `HashSet`. Estamos aqui usando o polimorfismo para nos proteger que mudanças de implementação venham nos obrigar a alterar muito código. Mais uma vez: *programe voltado a interface, e não à implementação*!

Esse é um **excelente** exemplo de bom uso de interfaces, afinal, de que importa como a coleção funciona? O que queremos é uma coleção qualquer, isso é suficiente para os nossos propósitos! Nosso código está com **baixo acoplamento** em relação a estrutura de dados utilizada: podemos trocá-la facilmente.

Esse é um código extremamente elegante e flexível. Com o tempo você vai reparar que as pessoas tentam programar sempre se referindo a essas interfaces menos específicas, na medida do possível: métodos costumam receber e devolver `Collection` s, `List` s e `Set` s em vez de referenciar diretamente uma implementação. É o mesmo que ocorre com o uso de `InputStream` e

`OutputStream` : eles são o suficiente, não há porque forçar o uso de algo mais específico.

Obviamente, algumas vezes não conseguimos trabalhar dessa forma e precisamos usar uma interface mais específica ou mesmo nos referir ao objeto pela sua implementação para poder chamar alguns métodos. Por exemplo, `TreeSet` tem mais métodos que os definidos em `Set` , assim como `LinkedList` em relação à `List` .

Dê um exemplo de um caso em que não poderíamos nos referir a uma coleção de elementos como `Collection` , mas necessariamente por interfaces mais específicas como `List` ou `Set` .

3. Faça testes com o `Map` , como visto nesse capítulo:

```
public class TestaMapa {

    public static void main(String[] args) {
        Conta c1 = new ContaCorrente();
        c1.deposita(10000);

        Conta c2 = new ContaCorrente();
        c2.deposita(3000);

        // cria o mapa
        Map mapaDeContas = new HashMap();

        // adiciona duas chaves e seus valores
        mapaDeContas.put("diretor", c1);
        mapaDeContas.put("gerente", c2);

        // qual a conta do diretor?
        Conta contaDoDiretor = (Conta) mapaDeContas.get("diretor");
        System.out.println(contaDoDiretor.getSaldo());
    }
}
```

Depois, altere o código para usar o *generics* e não haver a necessidade do casting, além da garantia de que nosso mapa estará seguro em relação a tipagem usada.

Você pode utilizar o *quickfix* do Eclipse para que ele conserte isso para você: na linha em que você está chamando o `put` , use o `ctrl + 1` . Depois de mais um quickfix (descubra!) seu código deve ficar como segue:

```
// cria o mapa
Map<String, Conta> mapaDeContas = new HashMap<>();
```

Que opção do `ctrl + 1` você escolheu para que ele adicionasse o *generics* para você?

4. (opcional) Assim como no exercício 1, crie uma comparação entre `ArrayList` e `LinkedList` , para ver qual é a mais rápida para se adicionar elementos na primeira posição (`list.add(0, elemento)`), como por exemplo:

```
public class TestaPerformanceDeAdicionarNaPrimeiraPosicao {
    public static void main(String[] args) {
        System.out.println("Iniciando...");
```

```
        long inicio = System.currentTimeMillis();

        // trocar depois por ArrayList
        List<Integer> teste = new LinkedList<>();

        for (int i = 0; i < 30000; i++) {
            teste.add(0, i);
        }

        long fim = System.currentTimeMillis();
        double tempo = (fim - inicio) / 1000.0;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Seguindo o mesmo raciocínio, você pode ver qual é a mais rápida para se percorrer usando o `get(indice)` (sabemos que o correto seria utilizar o *enhanced for* ou o `Iterator`). Para isso, insira 30 mil elementos e depois percorra-os usando cada implementação de `List` .

Perceba que aqui o nosso intuito não é que você aprenda qual é o mais rápido, o importante é perceber que podemos tirar proveito do polimorfismo para nos comprometer apenas com a interface. Depois, quando necessário, podemos trocar e escolher uma implementação mais adequada as nossas necessidades.

Qual das duas listas foi mais rápida para adicionar elementos à primeira posição?

5. (opcional) Crie a classe `Banco` (caso não tenha sido criada anteriormente) no pacote `br.com.caelum.contas.modelo` que possui uma `List` de `Conta` chamada `contas` . Repare que numa lista de `Conta` , você pode colocar tanto `ContaCorrente` quanto `ContaPoupanca` por causa do polimorfismo.

   Crie um método `void adiciona(Conta c)` , um método `Conta pega(int x)` e outro `int pegaQuantidadeDeContas()` . Basta usar a sua lista e delegar essas chamadas para os métodos e coleções que estudamos.

   Como ficou a classe `Banco` ?

6. (opcional) No `Banco` , crie um método `Conta buscaPorTitular(String nome)` que procura por uma `Conta` cujo `titular` seja `equals` ao `nomeDoTitular` dado.

   Você pode implementar esse método com um `for` na sua lista de `Conta` , porém não tem uma performance eficiente.

   Adicionando um atributo privado do tipo `Map<String, Conta>` terá um impacto significativo. Toda vez que o método `adiciona(Conta c)` for invocado, você deve invocar `.put(c.getTitular(), c)` no seu mapa. Dessa maneira, quando alguém invocar o método `Conta buscaPorTitular(String nomeDoTitular)` , basta você fazer o `get` no seu mapa, passando

`nomeDoTitular` como argumento!

Note, apenas, que isso é só um exercício! Dessa forma você não poderá ter dois clientes com o mesmo nome nesse banco, o que sabemos que não é legal.

Como ficaria sua classe `Banco` com esse `Map` ?

7. (opcional, avançado) Crie o método `hashCode` para a sua conta, de forma que ele respeite o `equals` de que duas contas são `equals` quando tem o mesmo número e agência. Felizmente para nós, o próprio Eclipse já vem com um criador de `equals` e `hashCode` que os faz de forma consistente.

   Na classe `Conta` , use o `ctrl + 3` e comece a escrever *hashCode* para achar a opção de gerá-los. Então, selecione os atributos `numero` e `agencia` e mande gerar o `hashCode` e o `equals` .

   Como ficou o código gerado?

8. (opcional, avançado) Crie uma classe de teste e verifique se sua classe `Conta` funciona agora corretamente em um `HashSet` , isto é, que ela não guarda contas com número e agência repetidos. Depois, remova o método `hashCode` . Continua funcionando?

   Dominar o uso e o funcionamento do `hashCode` é fundamental para o bom programador.

15.16 DESAFIOS

1. Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `TreeSet` . Como ficou seu código?

2. Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `ArrayList` . Como ficou seu código?

15.17 PARA SABER MAIS: COMPARATORS, CLASSES ANÔNIMAS, JAVA 8 E O LAMBDA

E se precisarmos ordernar uma lista com outro critério de comparação? Se precisarmos alterar a própria classe e mudar seu método `compareTo`, teremos apenas uma forma de comparação por vez. Precisamos de mais!

É possível definir outros critérios de ordenação usando a interface do `java.util` chamada `Comparator`. Existe um método `sort` em `Collections` que recebe, além da `List`, um `Comparator` definindo um critério de ordenação específico. É possível ter vários `Comparator` s com critérios diferentes para usar quando for necessário.

Vamos criar um `Comparator` que serve para ordernar `Strings` de acordo com seu tamanho.

```
class ComparadorPorTamanho implements Comparator<String> {
    public int compare(String s1, String s2) {
        if(s1.length() < s2.length())
            return -1;
        if(s2.length() < s1.length())
            return 1;
        return 0;
    }
}
```

Repare que, diferente de `Comparable`, o método aqui se chama `compare` e recebe dois argumentos, já que quem o implementa não é o próprio objeto.

Podemos deixá-lo mais curto, tomando proveito do método estático auxiliar `Integer.compare` que compara dois inteiros:

```
class ComparadorPorTamanho implements Comparator<String> {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
}
```

Depois, dentro do nosso código, teríamos uma chamada a `Collections.sort` passando o comparador também:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");

// invocando o sort passando o comparador
ComparadorPorTamanho comparador = new ComparadorPorTamanho();
Collections.sort(lista, comparador);

System.out.println(lista);
```

Como a variável temporária `comparador` é utilizada apenas aí, é comum escrevermos diretamente `Collections.sort(lista, new ComparadorPorTamanho())`.

Escrevendo um Comparator com classe anônima

Repare que a classe `ComparadorPorTamanho` é bem pequena. É comum haver a necessidade de criar vários critérios de comparação, e muitas vezes eles são utilizados apenas num único ponto do nosso programa.

Há uma forma de escrever essa classe e instanciá-la numa única instrução. Você faz isso dando `new` em `Comparator`. Mas como, se dissemos que uma interface não pode ser instanciada? Realmente `new Comparator()` não compila. Mas vai compilar se você abrir chaves e implementar tudo o que é necessário. Veja o código:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");


Comparator<String> comparador = new Comparator<String>() {
   public int compare(String s1, String s2) {
      return Integer.compare(s1.length(), s2.length());
   }
};
Collections.sort(lista, comparador);

System.out.println(lista);
```

A sintaxe é realmente esdrúxula! Numa única linha nós definimos uma classe e a instanciamos! Uma classe que nem mesmo nome tem. Por esse motivo o recurso é chamado de classe anônima. Ele aparece com certa frequência, em especial para não precisar implementar interfaces que o código dos métodos seriam muito curtos e não-reutilizáveis.

Há ainda como diminuir ainda mais o código, evitando a criação da variável temporária `comparador` e instanciando a interface dentro da invocação para o `sort`:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");

Collections.sort(lista, new Comparator<String>() {
   public int compare(String s1, String s2) {
      return Integer.compare(s1.length(), s2.length());
   }
});

System.out.println(lista);
```

Escrevendo um Comparator com lambda no Java 8

Você pode fazer o download do Java 8 aqui:

https://jdk8.java.net/download.html

O Eclipse já possui atualizações para compatibilidade com a nova versão, mas ele pode ser relativamente instável. Você pode utilizar a linha de comando, como fizemos no começo do curso, para esses testes, caso ache necessário.

A partir dessa nova versão do Java há uma forma mais simples de obter esse mesmo `Comparator`. Repare:

```
Collections.sort(lista, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

O código `(s1, s2) -> Integer.compare(s1.length(), l2.length())` gerará uma instância de `Comparator` que o `compare` devolve `Integer.compare(s1.length, l2.length)`. Até mesmo o `return` não é necessário, já que só temos uma instrução após o `->`. Esse é o recurso de lambda do Java 8.

Uma outra novidade do Java 8 é a possibilidade de declarar métodos concretos dentro de uma interface, os chamados *default methods*. Até o Java 7 não existia `sort` em listas. Colocar um novo método abstrato em uma interface pode ter consequências drásticas: todo mundo que a implementava para de compilar! Mas colocar um método default não tem esse mesmo impacto devastador, já que as classes que implementama interface 'herdam' esse método. Então você pode fazer:

```
lista.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Há outros métodos nas coleções que utilizam do lambda para serem mais sucintos.

Um deles é o `forEach`. Você pode fazer `lista.forEach(s -> System.out.println(s))`.

O `removeIf` é outro deles. Por exemplo, podemos escrever `lista.removeIf(c -> c.getSaldo() < 0)`. O `removeIf` recebe como argumento um objeto que implemente a interface `Predicate`, que possui apenas um método, que recebe um element e devolve `boolean`. Por possuir apenas um método abstrato também chamamos essa interface é uma interface funcional. O mesmo ocorre ao invocar o `forEach`, que recebe um argumento que implementa a interface funcional `Consumer`.

Mais? Method references, streams e collectors

Trabalhar com lambdas no Java 8 vai muito além. Há diversos detalhes e recursos que não veremos nesse primeiro curso. Caso tenha curiosidade e queira saber mais, veja no blog:

http://blog.caelum.com.br/o-minimo-que-voce-deve-saber-de-java-8/

CAPÍTULO 16

PACOTE JAVA.IO

"A benevolência é sobretudo um vício do orgulho e não uma virtude da alma." -- Doantien Alphonse François (Marquês de Sade)

Ao término desse capítulo, você será capaz de:

- usar as classes wrappers (como `Integer`) e boxing;
- ler e escrever bytes, caracteres e Strings de/para a entrada e saída padrão;
- ler e escrever bytes, caracteres e Strings de/para arquivos;
- utilizar buffers para agilizar a leitura e escrita através de fluxos;
- usar Scanner e PrintStream.

16.1 CONHECENDO UMA API

Vamos passar a conhecer APIs do Java. `java.io e java.util` possuem as classes que você mais comumente vai usar, não importando se seu aplicativo é desktop, web, ou mesmo para celulares.

Apesar de ser importante conhecer nomes e métodos das classes mais utilizadas, o interessante aqui é que você enxergue que todos os conceitos previamente estudados são aplicados a toda hora nas classes da biblioteca padrão.

Não se preocupe em decorar nomes. Atenha-se em entender como essas classes estão relacionadas e como elas estão tirando proveito do uso de interfaces, polimorfismo, classes abstratas e encapsulamento. Lembre-se de estar com a documentação (javadoc) aberta durante o contato com esses pacotes.

Veremos também threads e sockets em capítulos posteriores, que ajudarão a condensar nosso conhecimento, tendo em vista que no exercício de sockets utilizaremos todos conceitos aprendidos, juntamente com as várias APIs.

16.2 ORIENTAÇÃO A OBJETOS NO JAVA.IO

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada e saída de dados (conhecido como **io**) é orientada a objetos e usa os principais conceitos mostrados até agora: interfaces, classes abstratas e polimorfismo.

A ideia atrás do polimorfismo no pacote `java.io` é de utilizar fluxos de entrada (`InputStream`) e de saída (`OutputStream`) para toda e qualquer operação, seja ela relativa a um **arquivo**, a um campo **blob** do banco de dados, a uma conexão remota via **sockets**, ou até mesmo às **entrada** e **saída padrão** de um programa (normalmente o teclado e o console).

As classes abstratas `InputStream` e `OutputStream` definem, respectivamente, o comportamento padrão dos fluxos em Java: em um fluxo de entrada, é possível ler bytes e, no fluxo de saída, escrever bytes.

A grande vantagem dessa abstração pode ser mostrada em um método qualquer que utiliza um `OutputStream` recebido como argumento para escrever em um fluxo de saída. Para onde o método está escrevendo? Não se sabe e não importa: quando o sistema precisar escrever em um arquivo ou em uma socket, basta chamar o mesmo método, já que ele aceita qualquer filha de `OutputStream` !

16.3 INPUTSTREAM, INPUTSTREAMREADER E BUFFEREDREADER

Para ler um `byte` de um arquivo, vamos usar o leitor de arquivo, o `FileInputStream` . Para um `FileInputStream` conseguir ler um byte, ele precisa saber de onde ele deverá ler. Essa informação é tão importante que quem escreveu essa classe obriga você a passar o nome do arquivo pelo construtor: sem isso o objeto não pode ser construído.

```
class TestaEntrada {
```

```java
    public static void main(String[] args) throws IOException {
        InputStream is = new FileInputStream("arquivo.txt");
        int b = is.read();
    }
}
```

A classe `InputStream` é abstrata e `FileInputStream` uma de suas filhas concretas. `FileInputStream` vai procurar o arquivo no diretório em que a JVM fora invocada (no caso do Eclipse, vai ser a partir de dentro do diretório do projeto). Alternativamente você pode usar um caminho absoluto.

Quando trabalhamos com `java.io` , diversos métodos lançam `IOException` , que é uma exception do tipo checked - o que nos obriga a tratá-la ou declará-la. Nos exemplos aqui, estamos declarando `IOException` através da clausula `throws` do `main` apenas para facilitar o exemplo. Caso a exception ocorra, a JVM vai parar, mostrando a stacktrace. Esta não é uma boa prática em uma aplicação real: trate suas exceptions para sua aplicação poder abortar elegantemente.

`InputStream` tem diversas outras filhas, como `ObjectInputStream` , `AudioInputStream` , `ByteArrayInputStream` , entre outras.

Para recuperar um caractere, precisamos traduzir os bytes com o encoding dado para o respectivo código unicode, isso pode usar um ou mais bytes. Escrever esse decodificador é muito complicado, quem faz isso por você é a classe `InputStreamReader` .

```java
class TestaEntrada {
    public static void main(String[] args) throws IOException {
        InputStream is = new FileInputStream("arquivo.txt");
        InputStreamReader isr = new InputStreamReader(is);
        int c = isr.read();
    }
}
```

O construtor de `InputStreamReader` pode receber o encoding a ser utilizado como parâmetro, se desejado, tal como `UTF-8` ou `ISO-8859-1` .

Encodings

Devido a grande quantidade de aplicativos internacionalizados de hoje em dia, é imprescindível que um bom programador entenda bem o que são os character encodings e o Unicode. O blog da Caelum possui um bom artigo a respeito:

http://blog.caelum.com.br/2006/10/22/entendendo-unicode-e-os-character-encodings/

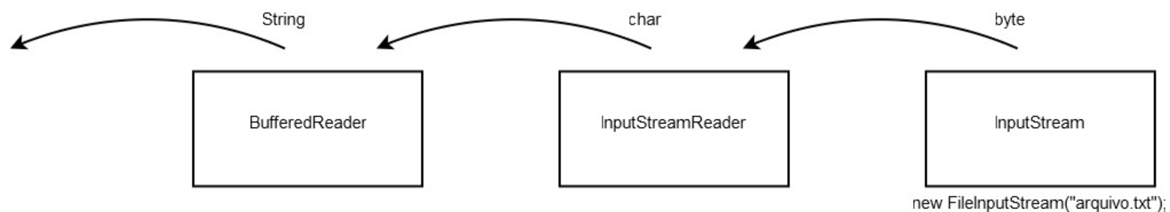
`InputStreamReader` é filha da classe abstrata `Reader` , que possui diversas outras filhas - são classes que manipulam chars.

Apesar da classe abstrata `Reader` já ajudar no trabalho de manipulação de caracteres, ainda seria difícil pegar uma `String`. A classe `BufferedReader` é um `Reader` que recebe outro `Reader` pelo construtor e concatena os diversos chars para formar uma `String` através do método `readLine`:

```
class TestaEntrada {
    public static void main(String[] args) throws IOException {
        InputStream is = new FileInputStream("arquivo.txt");
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
    }
}
```

Como o próprio nome diz, essa classe lê do `Reader` por pedaços (usando o buffer) para evitar realizar muitas chamadas ao sistema operacional. Você pode até configurar o tamanho do buffer pelo construtor.

É essa a composição de classes que está acontecendo:



Esse padrão de composição é bastante utilizado e conhecido. É o **Decorator Pattern**.

Aqui, lemos apenas a primeira linha do arquivo. O método `readLine` devolve a linha que foi lida e muda o cursor para a próxima linha. Caso ele chegue ao fim do `Reader` (no nosso caso, fim do arquivo), ele vai devolver `null`. Então, com um simples laço, podemos ler o arquivo por inteiro:

```
class TestaEntrada {
    public static void main(String[] args) throws IOException {
        InputStream is = new FileInputStream("arquivo.txt");
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        String s = br.readLine(); // primeira linha

        while (s != null) {
            System.out.println(s);
            s = br.readLine();
        }

        br.close();
    }
}
```

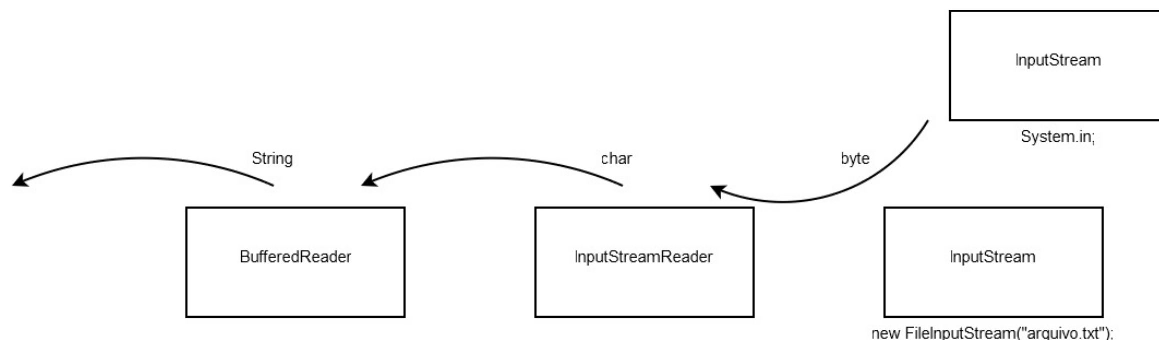
16.4 LENDO STRINGS DO TECLADO

Com um passe de mágica, passamos a ler do teclado em vez de um arquivo, utilizando o `System.in`, que é uma referência a um `InputStream` o qual, por sua vez, lê da entrada padrão.

```
class TestaEntrada {
    public static void main(String[] args) throws IOException {
        InputStream is = System.in;
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();

        while (s != null) {
            System.out.println(s);
            s = br.readLine();
        }
    }
}
```

Apenas modificamos a quem a variável `is` está se referindo. Podemos receber argumentos do tipo `InputStream` e ter esse tipo de abstração: não importa exatamente de onde estamos lendo esse punhado de bytes, desde que a gente receba a informação que estamos querendo. Como na figura:

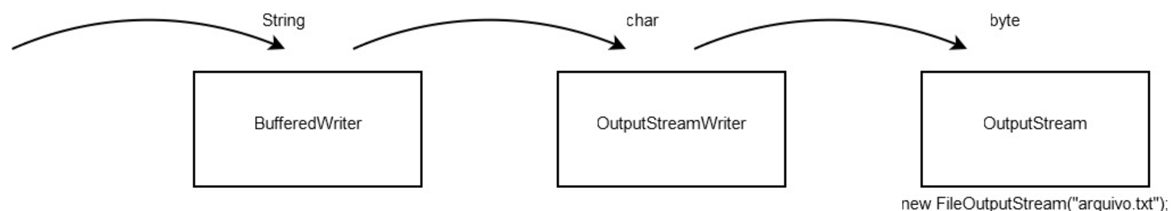


Repare que a ponta da direita poderia ser qualquer `InputStream`, seja `ObjectInputStream`, `AudioInputStream`, `ByteArrayInputStream`, ou a nossa `FileInputStream`. Polimorfismo! Ou você mesmo pode criar uma filha de `InputStream`, se desejar.

Por isso é muito comum métodos receberem e retornarem `InputStream`, em vez de suas filhas específicas. Com isso, elas desacoplam as informações e escondem a implementação, facilitando a mudança e manutenção do código. Repare que isso vai ao encontro de tudo o que aprendemos durante os capítulos que apresentaram classes abstratas, interfaces, polimorfismo e encapsulamento.

16.5 A ANALOGIA PARA A ESCRITA: OUTPUTSTREAM

Como você pode imaginar, escrever em um arquivo é o mesmo processo:




```
class TestaSaida {
    public static void main(String[] args) throws IOException {
        OutputStream os = new FileOutputStream("saida.txt");
        OutputStreamWriter osw = new OutputStreamWriter(os);
        BufferedWriter bw = new BufferedWriter(osw);

        bw.write("caelum");

        bw.close();
    }
}
```

Lembre-se de dar *refresh* (clique da direita no nome do projeto, refresh) no seu projeto do Eclipse para que o arquivo criado apareça. O `FileOutputStream` pode receber um booleano como segundo parâmetro, para indicar se você quer reescrever o arquivo ou manter o que já estava escrito (`append`).

O método `write` do `BufferedWriter` não insere o(s) caractere(s) de quebra de linha. Para isso, você pode chamar o método `newLine` .

FECHANDO O ARQUIVO COM O FINALLY E O TRY-WITH-RESOURCES

É importante sempre fechar o arquivo. Você pode fazer isso chamando diretamente o método `close` do `FileInputStream` / `OutputStream` , ou ainda chamando o `close` do `BufferedReader` / `Writer` . Nesse último caso, o `close` será cascateado para os objetos os quais o `BufferedReader` / `Writer` utiliza para realizar a leitura/escrita, além dele fazer o **flush** dos buffers no caso da escrita.

É comum e fundamental que o `close` esteja dentro de um bloco `finally` . Se um arquivo for esquecido aberto e a referência para ele for perdida, pode ser que ele seja fechado pelo *garbage collector*, que veremos mais a frente, por causa do `finalize` . Mas não é bom você se prender a isso. Se você esquecer de fechar o arquivo, no caso de um programa minúsculo como esse, o programa vai terminar antes que o tal do garbage collector te ajude, resultando em um arquivo não escrito (os bytes ficaram no buffer do `BufferedWriter`). Problemas similares podem acontecer com leitores que não forem fechados.

No Java 7 há a estrutura *try-with-resources*, que já fará o `finally` cuidar dos recursos declarados dentro do `try()` , invocando `close` . Pra isso, os recursos devem implementar a interface `java.lang.AutoCloseable` , que é o caso dos Readers, Writers e Streams estudados aqui:

```
try (BufferedReader br = new BufferedReader(new File("arquivo.txt"))) {
  // com exceção ou não, o close() do br sera invocado
}
```

16.6 UMA MANEIRA MAIS FÁCIL: SCANNER E PRINTSTREAM

A partir do Java 5, temos a classe `java.util.Scanner` , que facilita bastante o trabalho de ler de um `InputStream` . Além disso, a classe `PrintStream` possui um construtor que já recebe o nome de um arquivo como argumento. Dessa forma, a leitura do teclado com saída para um arquivo ficou muito simples:

```
Scanner s = new Scanner(System.in);
PrintStream ps = new PrintStream("arquivo.txt");
while (s.hasNextLine()) {
    ps.println(s.nextLine());
}
```

Nenhum dos métodos lança `IOException` : `PrintStream` lança `FileNotFoundException` se você o construir passando uma `String` . Essa exceção é filha de `IOException` e indica que o arquivo não foi encontrado. O `Scanner` considerará que chegou ao fim se uma `IOException` for lançada, mas

o `PrintStream` simplesmente engole exceptions desse tipo. Ambos possuem métodos para você verificar se algum problema ocorreu.

A classe `Scanner` é do pacote `java.util`. Ela possui métodos muito úteis para trabalhar com Strings, em especial, diversos métodos já preparados para pegar números e palavras já formatadas através de expressões regulares. Fica fácil parsear um arquivo com qualquer formato dado.

SYSTEM.OUT

Como vimos no capítulo passado, o atributo `out` da classe `System` é do tipo `PrintStream` (e, portanto, é um `OutputStream`).

EOF

Quando rodar sua aplicação, para encerrar a entrada de dados do teclado, é necessário enviarmos um sinal de fim de stream. É o famoso **EOF**, isto é, *end of file*.

No Linux/Mac/Solaris/Unix você faz isso com o `ctrl + D`. No Windows, use o `ctrl + Z`.

16.7 UM POUCO MAIS...

- Existem duas classes chamadas `java.io.FileReader` e `java.io.FileWriter`. Elas são atalhos para a leitura e escrita de arquivos.
- O `do { .. } while(condicao);` é uma alternativa para se construir um laço. Pesquise-o e utilize-o no código para ler um arquivo, ele vai ficar mais sucinto (você não precisará ler a primeira linha fora do laço).

Você pode também fazer o curso data dessa apostila na Caelum



Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

16.8 INTEGER E CLASSES WRAPPERS (BOX)

Anteriormente, vimos que conseguimos ler e escrever dados em um arquivo no Java utilizando a classe `Scanner` . Por padrão, quando fazemos essas operações, estamos trabalhando sempre com os dados em forma de `String` . Mas e se precisássemos ler ou escrever números inteiros em um arquivo? Como faríamos para transformar esses números em `String` e vice-versa?

Cuidado! Usamos aqui o termo "transformar", porém o que ocorre não é uma transformação entre os tipos e sim uma forma de conseguirmos um `String` dado um `int` e vice-versa. O jeito mais simples de transformar um número em `String` é concatená-lo da seguinte maneira:

```
int i = 100;
String s = "" + i;
System.out.println(s);

double d = 1.2;
String s2 = "" + d;
System.out.println(s2);
```

Para formatar o número de uma maneira diferente, com vírgula e número de casas decimais devemos utilizar outras classes de ajuda (`NumberFormat` , `Formatter`).

Para transformar uma `String` em número, utilizamos as classes de ajuda para os tipos primitivos correspondentes. Por exemplo, para transformar a `String s` em um número inteiro utilizamos o método estático da classe `Integer` :

```
String s = "101";
int i = Integer.parseInt(s);
```

As classes `Double` , `Short` , `Long` , `Float` etc contêm o mesmo tipo de método, como `parseDouble` e `parseFloat` que retornam um `double` e `float` respectivamente.

Essas classes também são muito utilizadas para fazer o **wrapping** (embrulho) de tipos primitivos

como objetos, pois referências e tipos primitivos são incompatíveis. Imagine que precisamos passar como argumento um inteiro para o nosso guardador de objetos. Um inteiro não é um `Object` , como fazer?

```
int i = 5;
Integer x = new Integer(i);
guardador.adiciona(x);
```

E, dado um `Integer` , podemos pegar o `int` que está dentro dele (desembrulhá-lo):

```
int i = 5;
Integer x = new Integer(i);
int numeroDeVolta = x.intValue();
```

16.9 AUTOBOXING NO JAVA 5.0

Esse processo de wrapping e unwrapping é entediante. O Java 5.0 em diante traz um recurso chamado de **autoboxing**, que faz isso sozinho para você, custando legibilidade:

```
Integer x = 5;
int y = x;
```

No Java 1.4 esse código é inválido. No Java 5.0 em diante ele compila perfeitamente. É importante ressaltar que isso não quer dizer que tipos primitivos e referências sejam do mesmo tipo, isso é simplesmente um "açúcar sintático" (*syntax sugar*) para facilitar a codificação.

Você pode fazer todos os tipos de operações matemáticas com os wrappers, porém corre o risco de tomar um `NullPointerException` .

Você pode fazer o autoboxing diretamente para `Object` também, possibilitando passar um tipo primitivo para um método que receber `Object` como argumento:

```
Object o = 5;
```

16.10 PARA SABER MAIS: JAVA.LANG.MATH

Na classe `Math` , existe uma série de métodos estáticos que fazem operações com números como, por exemplo, arredondar(`round`), tirar o valor absoluto (`abs`), tirar a raiz(`sqrt`), calcular o seno(`sin`) e outros.

```
double d = 4.6;
long i = Math.round(d);

int x = -4;
int y = Math.abs(x);
```

Consulte a documentação para ver a grande quantidade de métodos diferentes.

No Java 5.0, podemos tirar proveito do `import static` aqui:

```
import static java.lang.Math.*;
```

Isso elimina a necessidade de usar o nome da classe, sob o custo de legibilidade:

```
double d = 4.6;
long i = round(d);
int x = -4;
int y = abs(x);
```


16.11 EXERCÍCIOS: JAVA I/O

Vamos salvar as contas cadastradas em um arquivo para não precisar ficar adicionando as contas a todo momento.

1. Na classe `ManipuladorDeContas` , crie o método `salvaDados` que recebe um `Evento` de onde obteremos a lista de contas.

   Dica: a classe `Evento` possui o método `getLista("listaContas")` que vai lhe ajudar neste item.

2. Para não colocarmos todo o código de gerenciamento de arquivos dentro da classe `ManipuladorDeContas` , vamos criar uma nova classe cuja responsabilidade será lidar com a escrita / leitura de arquivos.

   Crie a classe `RepositorioDeContas` dentro do pacote `br.com.caelum.contas` e declare o método `salva` que deverá receber a lista de contas a serem guardadas. Neste método você deve percorrer a lista de contas e salvá-las separando as informações de `tipo` , `numero` , `agencia` , `titular` e `saldo` com vírgulas.

   Dica: Você vai precisar da classe java.io.PrintStream para fazer este item.

   O compilador vai reclamar que você não está tratando algumas exceções (como

`java.io.FileNotFoundException`). Utilize o devido `try` / `catch` e relance a exceção como `RuntimeException` . Utilize o *quick fix* do Eclipse para facilitar (**ctrl + 1**).

Vale lembrar que deixar todas as exceptions passarem despercebidas não é uma boa prática! Você pode usar aqui, pois estamos focando apenas no aprendizado da utilização do `java.io` .

Quando trabalhamos com recursos que falam com a parte externa à nossa aplicação, é preciso que avisemos quando acabarmos de usar esses recursos. Por isso, é **importantíssimo** lembrar de fechar os canais com o exterior que abrimos utilizando o método `close` !

3. Voltando à classe `ManipuladorDeContas` , vamos completar o método `salvaDados` para que utilize a nossa nova classe `RepositorioDeContas` criada:

   - No corpo do método, crie uma lista de contas e atribua à ela o retetorno do método `getLista` da classe `Evento` :

     ```
     List<Conta> contas = evento.getLista("listaContas");
     ```

     Dica: Aqui você vai precisar invocar o método `salva` da classe `RepositorioDeContas` .

     Rode sua aplicação, cadastre algumas contas e veja se aparece um arquivo chamado `contas.txt` dentro do diretório `src` de seu projeto. Talvez seja necessário dar um F5 nele para que o arquivo apareça.

4. (Opcional, Difícil) Vamos fazer com que além de salvar os dados em um arquivo, nossa aplicação também consiga carregar as informações das contas para já exibir na tela. Para que a aplicação funcione, é necessário que a nossa classe `ManipuladorDeContas` possua um método chamado `carregaDados` que devolva uma `List<Conta>` . Vamos fazer o mesmo que anteriormente e encapsular a lógica de carregamento dentro da classe `RepositorioDeContas` :

   ```
   public List<Conta> carregaDados() {
       RepositorioDeContas repositorio = new RepositorioDeContas();
       return repositorio.carrega();
   }
   ```

   Faça o código referente ao método `carrega` que devolve uma `List` dentro da classe `RepositorioDeContas` utilizando a classe `Scanner` . Para obter os valores de cada atributo você pode utilizar o método `split` da `String` . Lembre-se que os atributos das contas são carregados na seguinte ordem: `tipo` , `numero` , `agencia` , `titular` e `saldo` . Exemplo:

   ```
   String linha = scanner.nextLine();
   String[] valores = linha.split(",");
   String tipo = valores[0];
   ```

   Além disso, a conta deve ser instanciada de acordo com o conteúdo do `tipo` obtido. Também fique atento pois os dados lidos virão sempre lidos em forma de `String` e para alguns atributos será

necessário transformar o dado nos tipos primitivos correspondentes. Por exemplo:

```
String numeroTexto = valores[1];
int numero = Integer.parseInt(numeroTexto);
```

5. (opcional) A classe `Scanner` é muito poderosa! Consulte seu javadoc para saber sobre o `delimiter` e os outros métodos `next`.

6. (opcional) Crie uma classe `TestaInteger` e vamos fazer comparações com Integers dentro do `main`:

```
Integer x1 = new Integer(10);
Integer x2 = new Integer(10);

if (x1 == x2) {
    System.out.println("igual");
} else {
    System.out.println("diferente");
}
```

   E se testarmos com o `equals`? O que podemos concluir?

7. (opcional) Um `double` não está sendo suficiente para guardar a quantidade de casas necessárias em uma aplicação. Preciso guardar um número decimal muito grande! O que poderia usar?

   O `double` também tem problemas de precisão ao fazer contas, por causa de arredondamentos da aritmética de ponto flutuante definido pela IEEE 754:

   http://en.wikipedia.org/wiki/IEEE_754

   Ele não deve ser usado se você precisa realmente de muita precisão (casos que envolvam dinheiro, por exemplo).

   Consulte a documentação, tente adivinhar onde você pode encontrar um tipo que te ajudaria para resolver esses casos e veja como é intuitivo! Qual é a classe que resolveria esses problemas?

   Lembre-se: no Java há muito já pronto. Seja na biblioteca padrão, seja em bibliotecas *open source* que você pode encontrar pela internet.

16.12 DISCUSSÃO EM AULA: DESIGN PATTERNS E O TEMPLATE METHOD

Aplicar bem os conceitos de orientação a objetos é sempre uma grande dúvida. Sempre queremos encapsular direito, favorecer a flexibilidade, desacoplar classes, escrever código elegante e de fácil manutenção. E ouvimos falar que a Orientação a Objetos ajuda em tudo isso.

Mas, onde usar herança de forma saudável? Como usar interfaces? Onde o polimorfismo me ajuda? Como encapsular direito? Classes abstratas são usadas em que situações?

Muitos anos atrás, grandes nomes do mundo da orientação a objetos perceberam que criar bons designs orientados a objetos era um grande desafio para muitas pessoas. Perceberam que muitos problemas de OO apareciam recorrentemente em vários projetos; e que as pessoas já tinham certas soluções para esses problemas clássicos (nem sempre muito elegantes).

O que fizeram foi criar **soluções padrões para problemas comuns** na orientação a objetos, e chamaram isso de **Design Patterns**, ou Padrões de Projeto. O conceito vinha da arquitetura onde era muito comum ter esse tipo de solução. E, em 1994, ganhou grande popularidade na computação com o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, um catálogo com várias dessas soluções escrito por Erich Gamma, Ralph Johnson, Richard Helm e John Vlissides (a Gangue dos Quatro, GoF).

Design Patterns tornou-se referência absoluta no bom uso da orientação a objetos. Outros padrões surgiram depois, em outras literaturas igualmente consagradas. O conhecimento dessas técnicas é imprescindível para o bom programador.

Discuta com o instrutor como Design Patterns ajudam a resolver problemas de modelagem em sistemas orientados a objetos. Veja como Design Patterns são aplicados em muitos lugares do próprio Java.

O instrutor comentará do `Template Method` e mostrará o código fonte do método read() da classe `java.io.InputStream`:

```
public int read(byte b[], int off, int len) throws IOException {
    if (b == null) {
       throw new NullPointerException();
    } else if (off < 0 || len < 0 || len > b.length - off) {
       throw new IndexOutOfBoundsException();
    } else if (len == 0) {
       return 0;
    }

    int c = read();
    if (c == -1) {
       return -1;
    }

    b[off] = (byte) c;

    int i = 1;
    try {
       for (; i < len ; i++) {
          c = read();
          if (c == -1) {
             break;
          }
          b[off + i] = (byte)c;
       }
    } catch (IOException ee) {
    }
    return i;
 }
```

Discuta em aula como esse método aplica conceitos importantes da orientação a objetos e promove flexibilidade e extensibilidade.

Capítulo 17

E AGORA?

"A primeira coisa a entender é que você não entende."--Soren Aabye Kierkegaard

Onde continuar ao terminar os exercícios de 'Java e Orientação a Objetos'? Aqui há um post com sugestões de como iniciar na carreira:

http://blog.caelum.com.br/como-posso-aprender-java-e-iniciar-na-carreira/

E você pode seguir nesses cursos e áreas:

17.1 WEB

Um dos principais usos do Java é rodar aplicações web. Entram aqui tecnologias como Servlets, JSPs e ferramentas famosas do mercado, como o Struts.

A Caelum oferece o curso FJ-21, onde você pode estudar os tópicos necessários para começar a trabalhar com Java na web usando as melhores práticas, design patterns e tecnologias do mercado. Essa apostila também está disponível para download.

17.2 PRATICANDO JAVA E USANDO BIBLIOTECAS

A melhor maneira para fixar tudo o que foi visto nos capítulos anteriores é planejar e montar pequenos sistemas. Pense na modelagem de suas classes, como e onde usar herança, polimorfismo,

encapsulamento e outros conceitos. Pratique o uso das APIs mais úteis do Java integrando-as ao seus sistemas.

O curso FJ-22 é um laboratório que além de demonstrar o uso diversas APIs e boas práticas, vai mostrar diversos design patterns e seus casos de uso.

17.3 GRUPOS DE USUÁRIOS

Diversos programadores com o mínimo ou máximo de conhecimento se reúnem online para a troca de dúvidas, informações e ideias sobre projetos, bibliotecas e muito mais. São os grupos de usuários de java.

Um dos mais importantes e conhecidos no Brasil é o GUJ:

http://www.guj.com.br

17.4 PRÓXIMOS CURSOS

O 'Falando em Java' não pára por aqui. A Caelum oferece uma grande variedade de cursos que você pode seguir. Alguns dos mais requisitados:

FJ-21: Java para desenvolvimento Web

FJ-22: Laboratório Java com Testes, JSF, Web Services e Design Patterns

FJ-25: Persistência com JPA, Hibernate e EJB lite

FJ-26: Laboratório Web com JSF e CDI

FJ-57: Desenvolvimento móvel com Google Android

FJ-91: Arquitetura e Design de Projetos Java

Consulte mais informações no nosso site e entre em contato conosco. Conheça nosso mapa de cursos:

http://www.caelum.com.br/mapa-dos-cursos/

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

CAPÍTULO 18

APÊNDICE - PROGRAMAÇÃO CONCORRENTE E THREADS

"O único lugar onde o sucesso vem antes do trabalho é no dicionário." -- *Albert Einstein*

Ao término desse capítulo, você será capaz de:

- executar tarefas simultaneamente;
- colocar tarefas para aguardar até que um determinado evento ocorra;
- entender o funcionamento do Garbage Collector.

18.1 THREADS

"Duas tarefas ao mesmo tempo"

Em várias situações, precisamos "rodar duas coisas ao mesmo tempo". Imagine um programa que gera um relatório muito grande em PDF. É um processo demorado e, para dar alguma satisfação para o usuário, queremos mostrar uma barra de progresso. Queremos então gerar o PDF e *ao mesmo tempo* atualizar a barrinha.

Pensando um pouco mais amplamente, quando usamos o computador também fazemos várias coisas simultaneamente: queremos navegar na internet e *ao mesmo tempo* ouvir música.

A necessidade de se fazer várias coisas simultaneamente, ao mesmo tempo, **paralelamente**, aparece frequentemente na computação. Para vários programas distintos, normalmente o próprio sistema operacional gerencia isso através de vários *processos* em paralelo.

Em um programa só (um processo só), se queremos executar coisas em paralelo, normalmente falamos de **Threads**.

Threads em Java

Em Java, usamos a classe `Thread` do pacote `java.lang` para criarmos *linhas de execução* paralelas. A classe `Thread` recebe como argumento um objeto com o código que desejamos rodar. Por exemplo, no programa de PDF e barra de progresso:

```
public class GeraPDF {
```

```
    public void rodar () {
        // lógica para gerar o pdf...
    }
}

public class BarraDeProgresso {
    public void rodar () {
        // mostra barra de progresso e vai atualizando ela...
    }
}
```

E, no método `main`, criamos os objetos e passamos para a classe `Thread`. O método `start` é responsável por iniciar a execução da `Thread`:

```
public class MeuPrograma {
    public static void main (String[] args) {

        GeraPDF gerapdf = new GeraPDF();
        Thread threadDoPdf = new Thread(gerapdf);
        threadDoPdf.start();

        BarraDeProgresso barraDeProgresso = new BarraDeProgresso();
        Thread threadDaBarra = new Thread(barraDeProgresso);
        threadDaBarra.start();

    }
}
```

O código acima, porém, não compilará. Como a classe `Thread` sabe que deve chamar o método `roda`? Como ela sabe que nome de método daremos e que ela deve chamar esse método especial? Falta na verdade um **contrato** entre as nossas classes a serem executadas e a classe `Thread`.

Esse contrato existe e é feito pela *interface* `Runnable`: devemos dizer que nossa classe é "executável" e que segue esse contrato. Na interface `Runnable`, há apenas um método chamado `run`. Basta implementá-lo, "assinar" o contrato e a classe `Thread` já saberá executar nossa classe.

```
public class GeraPDF implements Runnable {
    public void run () {
        // lógica para gerar o pdf...
    }
}

public class BarraDeProgresso implements Runnable {
    public void run () {
        // mostra barra de progresso e vai atualizando ela...
    }
}
```

A classe `Thread` recebe no construtor um objeto que **é um** `Runnable`, e seu método `start` chama o método `run` da nossa classe. Repare que a classe `Thread` não sabe qual é o tipo específico da nossa classe; para ela, basta saber que a classe segue o contrato estabelecido e possui o método `run`.

É o bom uso de interfaces, contratos e polimorfismo na prática!

ESTENDENDO A CLASSE THREAD

A classe `Thread` implementa `Runnable` . Então, você pode criar uma subclasse dela e reescrever o `run` que, na classe `Thread` , não faz nada:

```
public class GeraPDF extends Thread {
    public void run () {
        // ...
    }
}
```

E, como nossa classe **é uma** `Thread` , podemos usar o `start` diretamente:

```
GeraPDF gera = new GeraPDF();
gera.start();
```

Apesar de ser um código mais simples, você está usando herança apenas por "preguiça" (herdamos um monte de métodos mas usamos apenas o `run`), e não por polimorfismo, que seria a grande vantagem. Prefira implementar `Runnable` a herdar de `Thread` .

DORMINDO

Para que a thread atual durma basta chamar o método a seguir, por exemplo, para dormir 3 segundos:

```
javaThread.sleep(3 * 1000);
```

18.2 ESCALONADOR E TROCAS DE CONTEXTO

Veja a classe a seguir:

```
public class Programa implements Runnable {

    private int id;
    // colocar getter e setter pro atributo id

    public void run () {
        for (int i = 0; i < 10000; i++) {
            System.out.println("Programa " + id + " valor: " + i);
        }
    }
}
```

É uma classe que implementa `Runnable` e, no método `run`, apenas imprime dez mil números. Vamos usá-las duas vezes para criar duas threads e imprimir os números duas vezes simultaneamente:

```
public class Teste {
    public static void main(String[] args) {

        Programa p1 = new Programa();
        p1.setId(1);

        Thread t1 = new Thread(p1);
        t1.start();

        Programa p2 = new Programa();
        p2.setId(2);

        Thread t2 = new Thread(p2);
        t2.start();

    }
}
```

Se rodarmos esse programa, qual será a saída? De um a mil e depois de um a mil? Provavelmente não, senão seria sequencial. Ele imprimirá 0 de t1, 0 de t2, 1 de t1, 1 de t2, 2 de t1, 2 de t2 e etc? Exatamente intercalado?

Na verdade, não sabemos exatamente qual é a saída. Rode o programa várias vezes e observe: em cada execução a saída é um pouco diferente.

O problema é que no computador existe apenas um processador capaz de executar coisas. E quando queremos executar várias coisas ao mesmo tempo, e o processador só consegue fazer uma coisa de cada vez? Entra em cena o **escalonador de threads**.

O escalonador (**scheduler**), sabendo que apenas uma coisa pode ser executada de cada vez, pega todas as threads que precisam ser executadas e faz o processador ficar alternando a execução de cada uma delas. A ideia é executar um pouco de cada thread e fazer essa troca tão rapidamente que a impressão que fica é que as coisas estão sendo feitas ao mesmo tempo.

O escalonador é responsável por escolher qual a próxima thread a ser executada e fazer a **troca de contexto** (context switch). Ele primeiro salva o estado da execução da thread atual para depois poder retomar a execução da mesma. Aí ele restaura o estado da thread que vai ser executada e faz o processador continuar a execução desta. Depois de um certo tempo, esta thread é tirada do processador, seu estado (o contexto) é salvo e outra thread é colocada em execução. A *troca de contexto* é justamente as operações de salvar o contexto da thread atual e restaurar o da thread que vai ser executada em seguida.

Quando fazer a troca de contexto, por quanto tempo a thread vai rodar e qual vai ser a próxima thread a ser executada, são escolhas do escalonador. Nós não controlamos essas escolhas (embora possamos dar "dicas" ao escalonador). Por isso que nunca sabemos ao certo a ordem em que programas paralelos são executados.

Você pode pensar que é ruim não saber a ordem. Mas perceba que se a ordem importa para você, se é importante que determinada coisa seja feita antes de outra, então não estamos falando de execuções paralelas, mas sim de um programa sequencial normal (onde uma coisa é feita depois da outra, em uma sequência).

Todo esse processo é feito automaticamente pelo escalonador do Java (e, mais amplamente, pelo escalonador do sistema operacional). Para nós, programadores das threads, é como se as coisas estivessem sendo executadas ao mesmo tempo.

E EM MAIS DE UM PROCESSADOR?

A VM do Java e a maioria dos SOs modernos consegue fazer proveito de sistemas com vários processadores ou multi-core. A diferença é que agora temos mais de um processador executando coisas e teremos, sim, execuções verdadeiramente paralelas.

Mas o número de processos no SO e o número de Threads paralelas costumam ser tão grandes que, mesmo com vários processadores, temos as trocas de contexto. A diferença é que o escalonador tem dois ou mais processadores para executar suas threads. Mas dificilmente terá uma máquina com mais processadores que threads paralelas executando.

18.3 GARBAGE COLLECTOR

O **Garbage Collector** (coletor de lixo, lixeiro) funciona como uma `Thread` responsável por jogar fora todos os objetos que não estão sendo referenciados por nenhum outro objeto - seja de maneira direta ou indireta.

Considere o código:

```
Conta conta1 = new ContaCorrente();
Conta conta2 = new ContaCorrente();
```

Até este momento, sabemos que temos 2 objetos em memória. Aqui, o *Garbage Collector* não pode eliminar nenhum dos objetos, pois ainda tem alguém se referindo a eles de alguma forma.

Podemos, então, executar uma linha que nos faça perder a referência para um dos dois objetos criados, como, por exemplo, o seguinte código:

```
conta2 = conta1;
```

Quantos objetos temos em memória?

Perdemos a referência para um dos objetos que foram criados. Esse objeto já não é mais acessível. Temos, então, apenas um objeto em memória? Não podemos afirmar isso! Como o *Garbage Collector* é uma Thread, você não tem garantia de quando ele vai rodar. Você só sabe que, em algum momento no futuro, aquela memória vai ser liberada.

Algumas pessoas costumam atribuir `null` a uma variável, com o intuito de acelerar a passagem do *Garbage Collector* por aquele objeto:

```
for (int i = 0; i < 100; i++) {
    List x = new ArrayList();
    // faz algumas coisas com a arraylist
    x = null;
}
```

Isso rarissimamente é necessário. O *Garbage Collector* age apenas sobre objetos, nunca sobre variáveis. Nesse caso, a variável `x` não existirá mais a cada iteração, deixando a `ArrayList` criada sem nenhuma referência para ela.

SYSTEM.GC()

Você nunca consegue forçar o Garbage Collector, mas chamando o método estático `gc` da classe `System` , você está sugerindo que a Virtual Machine rode o Garbage Collector naquele momento. Se sua sugestão vai ser aceita ou não, isto depende de JVM para JVM, e você não tem garantias. Evite o uso deste método. Você não deve basear sua aplicação em quando o Garbage Collector vai rodar ou não.

FINALIZER

A classe `Object` define também um método `finalize` , que você pode reescrever. Esse método será chamado no instante antes do Garbage Collector coletar este objeto. Não é um destrutor, você não sabe em que momento ele será chamado. Algumas pessoas o utilizam para liberar recursos "caros" como conexões, threads e recursos nativos. Isso deve ser utilizado apenas por segurança: o ideal é liberar esses recursos o mais rápido possível, sem depender da passagem do Garbage Collector.

18.4 EXERCÍCIOS

1. Teste o exemplo deste capítulo para imprimir números em paralelo.

   Escreva a classe Programa:

```
public class Programa implements Runnable {

    private int id;
    // colocar getter e setter pro atributo id

    public void run () {
        for (int i = 0; i < 10000; i++) {
            System.out.println("Programa " + id + " valor: " + i);
        }
    }
}
```

   Escreva a classe de Teste:

```
public class Teste {
    public static void main(String[] args) {

        Programa p1 = new Programa();
        p1.setId(1);

        Thread t1 = new Thread(p1);
        t1.start();

        Programa p2 = new Programa();
        p2.setId(2);

        Thread t2 = new Thread(p2);
        t2.start();

    }
}
```

   Rode várias vezes a classe `Teste` e observe os diferentes resultados em cada execução. O que muda?

18.5 E AS CLASSES ANÔNIMAS?

É comum aparecer uma classe anônima junto com uma thread. Vimos como usá-la com o `Comparator`. Vamos ver como usar em um `Runnable`.

Considere um `Runnable` simples, que apenas manda imprimir algo na saída padrão:

```
public class Programa1 implements Runnable {
    public void run () {
        for (int i = 0; i < 10000; i++) {
            System.out.println("Programa 1 valor: " + i);
        }
    }
}
```

No seu `main`, você faz:

```
Runnable r = new Programa1();
Thread t = new Thread(r);
t.start();
```

Em vez de criar essa classe `Programa1`, podemos utilizar o recurso de classe anônima. Ela nos permite dar `new` numa interface, desde que implementemos seus métodos. Com isso, podemos colocar diretamente no `main`:

```
Runnable r = new Runnable() {
    public void run() {
        for(int i = 0; i < 10000; i++)
            System.out.println("programa 1 valor " + i);
    }
};
Thread t = new Thread(r);
t.start();
```

LIMITAÇÕES DAS CLASSES ANÔNIMAS

O uso de classes anônimas tem limitações. Não podemos declarar um construtor. Como estamos instanciando uma interface, então não conseguimos passar um parâmetro para ela. Como então passar o `id` como argumento? Você pode, de dentro de uma classe anônima, acessar atributos da classe dentro da qual foi declarada! Também pode acessar as variáveis locais do método, desde que eles sejam `final` .

E com lambda do Java 8?

Dá para ir mais longe com o Java 8, utilizando o lambda. Como `Runnable` é uma interface funcional (contém apenas um método abstrato), ela pode ser facilmente escrita dessa forma:

```
Runnable r = () -> {
    for(int i = 0; i < 10000; i++)
        System.out.println("programa 1 valor " + i);
};
Thread t = new Thread(r);
t.start();
```

A sintaxe pode ser um pouco estranha. Como não há parâmetros a serem recebidos pelo método `run` , usamos o `()` para indicar isso. Vale lembrar, mais uma vez, que no lambda não precisamos escrever o nome do método que estamos implementando, no nosso caso o `run` . Isso é possível pois existe apenas um método abstrato na interface.

Quer deixar o código mais enxuto ainda? Podemos passar o lambda diretamente para o construtor de `Thread` , sem criar uma variável temporária! E logo em seguida chamar o `start` :

```
new Thread(() -> {
    for(int i = 0; i < 10000; i++)
        System.out.println("programa 1 valor " + i);
}).start();
```

Obviamente o uso excessivo de lambdas e classes anônimas pode causar uma certa falta de legibilidade. Você deve lembrar que usamos esses recursos para escrever códigos mais legíveis, e não apenas para poupar algumas linhas de código. Caso nossa implementação do lambda venha a ser de várias linhas, é um forte sinal de que deveríamos ter uma classe a parte somente para ela.

CAPÍTULO 19

APÊNDICE - SOCKETS

"Olho por olho, e o mundo acabará cego."--Mohandas Gandhi

Conectando-se a máquinas remotas.

19.1 MOTIVAÇÃO: UMA API QUE USA OS CONCEITOS APRENDIDOS

Neste capítulo, você vai conhecer a API de **Sockets** do java pelo pacote `java.net`.

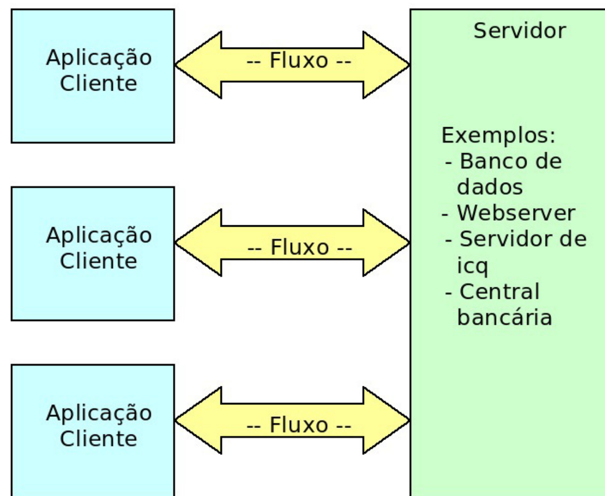
Mais útil que conhecer a API, é você perceber que estamos usando, aqui, todos os conceitos e bibliotecas aprendidas durante os outros capítulos. Repare, também, que é relativamente simples aprender a utilizar uma API, agora que temos todos os conceitos necessários para tal.

Lembre-se de fazer esse apêndice com o javadoc aberto ao seu lado.

19.2 PROTOCOLO

Da necessidade de dois computadores se comunicarem, surgiram diversos protocolos que permitissem tal troca de informação: o protocolo que vamos usar aqui é o **TCP** (*Transmission Control Protocol*).

Através do **TCP**, é possível criar um fluxo entre dois computadores - como é mostrado no diagrama abaixo:



É possível conectar mais de um cliente ao mesmo servidor, como é o caso de diversos banco de dados, servidores Web, etc.

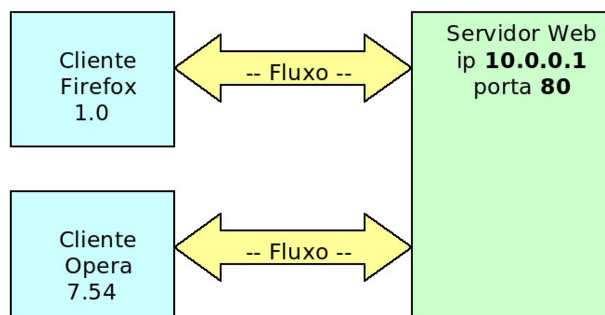
Ao escrever um programa em Java que se comunique com outra aplicação, não é necessário se preocupar com um nível tão baixo quanto o protocolo. As classes que trabalham com eles já foram disponibilizadas para serem usadas por nós no pacote `java.net`.

A vantagem de se usar TCP, em vez de criar nosso próprio protocolo de bytes, é que o TCP vai garantir a entrega dos pacotes que transferirmos e criar um protocolo base para isto é algo bem complicado.

19.3 PORTA

Acabamos de mencionar que diversos computadores podem se conectar a um só, mas, na realidade, é muito comum encontrar máquinas clientes com uma só conexão física. Então, como é possível se conectar a dois pontos? Como é possível ser conectado por diversos pontos?

Todas as aplicações que estão enviando e recebendo dados fazem isso através da mesma conexão física, mas o computador consegue discernir, durante a chegada de novos dados, quais informações pertencem a qual aplicação. Mas como?



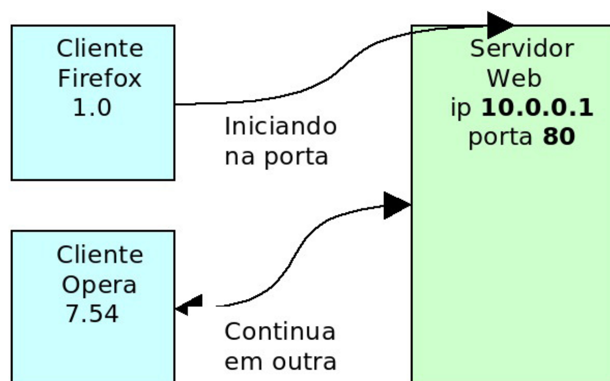
Assim como existe o **IP** para identificar uma máquina, a **porta** é a solução para identificar diversas aplicações em uma máquina. Esta porta é um número de 2 bytes, **varia de 0 a 65535**. Se todas as portas de uma máquina estiverem ocupadas, não é possível se conectar a ela enquanto nenhuma for liberada.

Ao configurar um servidor para rodar na porta 80 (padrão http), é possível se conectar a esse servidor através dessa porta que, junto com o ip, vai formar o endereço da aplicação. Por exemplo, o servidor web da *caelum.com.br* pode ser representado por: *caelum.com.br:80*

19.4 SOCKET

Mas se um cliente se conecta a um programa rodando na porta 80 de um servidor, enquanto ele não se desconectar dessa porta, será impossível que outra pessoa se conecte?

Acontece que, ao efetuar e aceitar a conexão, o servidor redireciona o cliente de uma porta para outra, liberando novamente sua porta inicial e permitindo que outros clientes se conectem novamente.



Em Java, isso deve ser feito através de threads e o processo de aceitar a conexão deve ser rodado o mais rápido possível.

19.5 SERVIDOR

Iniciando um modelo de servidor de chat, o serviço do computador que funciona como base deve, primeiro, abrir uma porta e ficar ouvindo até alguém tentar se conectar.

```
import java.net.*;

public class Servidor {
    public static void main(String[] args) throws IOException {

        ServerSocket servidor = new ServerSocket(12345);
        System.out.println("Porta 12345 aberta!");
        // a continuação do servidor deve ser escrita aqui

    }
}
```

Se o objeto for realmente criado, significa que a porta 12345 estava fechada e foi aberta. Se outro programa possui o controle desta porta neste instante, é normal que o nosso exemplo não funcione, pois ele não consegue utilizar uma porta que já está em uso.

Após abrir a porta, precisamos esperar por um cliente através do método `accept` da `ServerSocket`. Assim que um cliente se conectar, o programa continuará, por isso dizemos que esse método é *blocante*, segura a thread até que algo o notifique.

```
Socket cliente = servidor.accept();
System.out.println("Nova conexão com o cliente " +
    cliente.getInetAddress().getHostAddress()
); // imprime o ip do cliente
```

Por fim, basta ler todas as informações que o cliente nos enviar:

```
Scanner scanner = new Scanner(cliente.getInputStream());

while (scanner.hasNextLine()) {
    System.out.println(scanner.nextLine());
}
```

Fechamos as conexões, começando pelo fluxo:

```
in.close();
cliente.close();
servidor.close();
```

O resultado é a classe a seguir:

```
public class Servidor {
    public static void main(String[] args) throws IOException {
        ServerSocket servidor = new ServerSocket(12345);
        System.out.println("Porta 12345 aberta!");

        Socket cliente = servidor.accept();
        System.out.println("Nova conexão com o cliente " +
            cliente.getInetAddress().getHostAddress()
        );

        Scanner s = new Scanner(cliente.getInputStream());
```

```
        while (s.hasNextLine()) {
            System.out.println(s.nextLine());
        }

        s.close();
        servidor.close();
        cliente.close();
    }
}
```

19.6 CLIENTE

A nossa tarefa é criar um programa cliente que envie mensagens para o servidor... o cliente é ainda mais simples do que o servidor.

O código a seguir é a parte principal e tenta se conectar a um servidor no IP 127.0.0.1 (máquina local) e porta 12345:

```
    Socket cliente = new Socket("127.0.0.1",12345);
    System.out.println("O cliente se conectou ao servidor!");
```

Queremos ler os dados do cliente, da entrada padrão (teclado):

```
    Scanner teclado = new Scanner(System.in);
    while (teclado.hasNextLine()) {
        // lê a linha e faz algo com ela
    }
```

Basta ler as linhas que o usuário digitar através do buffer de entrada (`in`), e jogá-las no buffer de saída:

```
PrintStream saida = new PrintStream(cliente.getOutputStream());
Scanner teclado = new Scanner(System.in);
while (teclado.hasNextLine()) {
    saida.println(teclado.nextLine());
}
saida.close();
teclado.close();
```

Repare que usamos os conceito de `java.io` aqui novamente, para leitura do teclado e envio de mensagens para o servidor. Para as classes `Scanner` e `PrintStream` , tanto faz de onde que se lê ou escreve os dados: o importante é que esse stream seja um `InputStream` / `OutputStream` . É o poder das interfaces, do polimorfismo, aparecendo novamente.

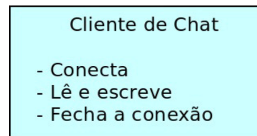
Nosso programa final:

```
public class Cliente {
    public static void main(String[] args)
                    throws UnknownHostException, IOException {
        Socket cliente = new Socket("127.0.0.1", 12345);
        System.out.println("O cliente se conectou ao servidor!");

        Scanner teclado = new Scanner(System.in);
        PrintStream saida = new PrintStream(cliente.getOutputStream());
```

```
        while (teclado.hasNextLine()) {
            saida.println(teclado.nextLine());
        }

        saida.close();
        teclado.close();
        cliente.close();
    }
}
```



Para testar o sistema, precisamos rodar primeiro o servidor e, logo depois, o cliente. Tudo o que for digitado no cliente será enviado para o servidor.

MULTITHREADING

Para que o servidor seja capaz de trabalhar com dois clientes ao mesmo tempo é necessário criar uma thread logo após executar o método `accept` .

A thread criada será responsável pelo tratamento dessa conexão, enquanto o laço do servidor disponibilizará a porta para uma nova conexão:

```
while (true) {

    Socket cliente = servidor.accept();

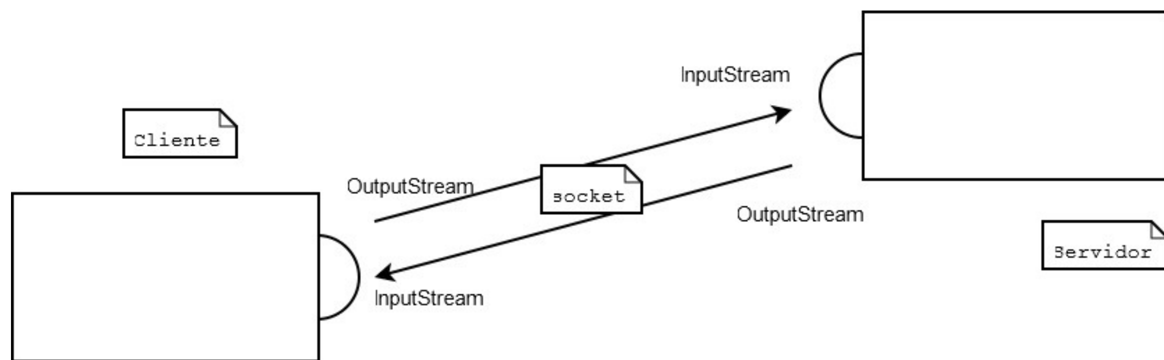
    // cria um objeto que vai tratar a conexão
    TratamentoClass tratamento = new TratamentoClass(cliente);

    // cria a thread em cima deste objeto
    Thread t = new Thread(tratamento);

    // inicia a thread
    t.start();

}
```

19.7 IMAGEM GERAL



A socket do cliente tem um `InputStream` , que recebe do `OutputStream` do servidor, e tem um `OutputStream` , que transfere tudo para o `InputStream` do servidor. Muito parecido com um telefone!

Repare que cliente e servidor são rótulos que indicam um estado. Um micro (ou melhor, uma JVM) pode ser servidor num caso, mas pode ser cliente em outro caso.

19.8 EXERCÍCIOS: SOCKETS

1. Crie um projeto `sockets` .

   Vamos fazer um pequeno sistema em que tudo que é digitado no micro cliente acaba aparecendo no micro servidor. Isto é, apenas uma comunicação unidirecional.

   Crie a classe `Servidor` como vimos nesse capítulo. Abuse dos recursos do Eclipse para não ter de escrever muito!

```
package br.com.caelum.chat;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
```

```java
public class Servidor {
    public static void main(String[] args) throws IOException {
        ServerSocket servidor = new ServerSocket(12345);
        System.out.println("Porta 12345 aberta!");

        Socket cliente = servidor.accept();
        System.out.println("Nova conexão com o cliente " +
            cliente.getInetAddress().getHostAddress());

        Scanner entrada = new Scanner(cliente.getInputStream());
        while (entrada.hasNextLine()) {
            System.out.println(entrada.nextLine());
        }

        entrada.close();
        servidor.close();
    }
}
```

2. Crie a classe `Cliente` como vista anteriormente:

```java
package br.com.caelum.chat;

import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class Cliente {
    public static void main(String[] args)
                throws UnknownHostException, IOException {
        Socket cliente = new Socket("127.0.0.1", 12345);
        System.out.println("O cliente se conectou ao servidor!");

        Scanner teclado = new Scanner(System.in);
        PrintStream saida = new PrintStream(cliente.getOutputStream());

        while (teclado.hasNextLine()) {
            saida.println(teclado.nextLine());
        }

        saida.close();
        teclado.close();
    }
}
```

   Utilize dos quick fixes e control espaço para os `import` s e o `throws` .

3. Rode a classe `Servidor` : repare no console do Eclipse que o programa fica esperando. Rode a classe `Cliente` : a conexão deve ser feita e o Eclipse deve mostrar os dois consoles para você (existe um pequeno ícone na view de Console para você alternar entre eles).

   Digite mensagens no cliente e veja se elas aparecem corretamente no servidor.

4. Teste seu programa com um colega do curso, usando comunicação remota entre as duas máquinas. Combinem entre si quem vai rodar o cliente e quem vai rodar o servidor. Quem for rodar o cliente

deve editar o IP na classe para indicar o endereço da outra máquina (verifique também se estão acessando a mesma porta).

> **Descobrindo o IP da máquina**
>
> No Windows, abra o console e digite *ipconfig* para saber qual é o seu IP. No Linux (ou no BSD, Mac, Solaris), vá no console e digite *ifconfig*.

5. (opcional) E se você quisesse, em vez de enviar tudo o que o cliente digitou, transferir um arquivo texto do micro do cliente para servidor? Seria difícil?

   Abuse do polimorfismo! Faça o cliente ler de um arquivo chamado `arquivo.txt` (crie-o!) e faça com que o servidor grave tudo que recebe num arquivo que chama `recebido.txt` .

19.9 DESAFIO: MÚLTIPLOS CLIENTES

Quando o servidor aceita um cliente com a chamada ao `accept` , ele poderia chamar novamente este método para aceitar um novo cliente. E, se queremos aceitar vários clientes, simultâneos, basta chamar o `accept` várias vezes e tratar cada cliente em sua própria `Thread` (senão o método `accept` não será invocado novamente!).

Um esboço de solução para a classe `Servidor` :

```
ServerSocket servidor = new ServerSocket(12345);

// servidor fica eternamente aceitando clientes...
while (true) {
    Socket cliente = servidor.accept();
    // dispara uma Thread que trata esse cliente e já espera o próximo
}
```

[TODO: seria legal essa solução parcial para apenas essa parte!]

19.10 DESAFIO: BROADCAST DAS MENSAGENS

Agora que vários clientes podem mandar mensagens, gostaríamos que os clientes recebessem as mensagens enviadas pelas outras pessoas. Ao invés do servidor simplesmente escrever as mensagens no console, ele deve mandar cada mensagem para todos os clientes conectados.

Precisamos manter uma lista de clientes conectados e, quando chegar uma mensagem (de qualquer cliente), percorremos essa lista e mandamos para todos.

Use um `List` para guardar os `PrintStream` s dos clientes. Logo depois que o servidor aceitar um

cliente novo, crie um `PrintStream` usando o `OutputStream` dele e adicione na lista. E, quando receber uma mensagem nova, envia para todos na lista.

Um esboço:

Adicionando na lista:

```
while (true) {
    Socket cliente = servidor.accept();
    this.lista.add(new PrintStream(cliente.getOutputStream()));

    // dispara uma Thread que trata esse cliente e já espera o próximo
}
```

Método que distribui as mensagens:

```
void distribuiMensagem(String msg) {
    for (PrintStream cliente : lista) {
        cliente.println(msg);
    }
}
```

Mas nosso cliente também recebe mensagens. Então precisamos fazer com que o Cliente, além de ler mensagens do teclado e enviar para o servidor, simultaneamente também possa receber mensagens de outros clientes enviadas pelo servidor.

Ou seja, precisamos de uma segunda Thread na classe `Cliente` que fica recebendo mensagens do `InputStream` do servidor e imprimindo no console.

Um esboço:

```
Scanner servidor = new Scanner(cliente.getInputStream());
while (servidor.hasNextLine()) {
    System.out.println(servidor.nextLine());
}
```

Lembre que você precisará de no mínimo 2 threads para o cliente e 2 para o servidor. Então provavelmente você vai ter que escrever 4 classes.

Melhorias possíveis:

- Faça com o a primeira linha enviada pelo cliente seja sempre o nick dele. E quando o servidor enviar a mensagem, faça ele enviar o nick de cada cliente antes da mensagem.
- E quando um cliente desconectar? Como retirá-lo da lista?
- É difícil fazer o envio de arquivos pelo nosso sistema de chats? Sabendo que a leitura de um arquivo é feita pelo `FileInputStream`, seria difícil mandar esse `InputStream` pelo `OutputStream` da conexão de rede?

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.]

19.11 SOLUÇÃO DO SISTEMA DE CHAT

Uma solução para o sistema de chat cliente-servidor com múltiplos clientes proposto nos desafios acima. Repare que a solução não está nem um pouco elegante: o `main` já faz tudo, além de não tratarmos as exceptions. O código visa apenas mostrar o uso de uma API. É uma péssima prática colocar toda a funcionalidade do seu programa no `main` e também de jogar exceções para trás.

Nesta listagem, faltam os devidos **imports**.

Primeiro, as duas classes para o cliente. Repare que a única mudança grande é a classe nova, Recebedor:

```
public class Cliente {
    public static void main(String[] args)
                throws UnknownHostException, IOException {
        // dispara cliente
        new Cliente("127.0.0.1", 12345).executa();
    }

    private String host;
    private int porta;

    public Cliente (String host, int porta) {
        this.host = host;
        this.porta = porta;
    }

    public void executa() throws UnknownHostException, IOException {
        Socket cliente = new Socket(this.host, this.porta);
        System.out.println("O cliente se conectou ao servidor!");

        // thread para receber mensagens do servidor
        Recebedor r = new Recebedor(cliente.getInputStream());
        new Thread(r).start();

        // lê msgs do teclado e manda pro servidor
```

```
        Scanner teclado = new Scanner(System.in);
        PrintStream saida = new PrintStream(cliente.getOutputStream());
        while (teclado.hasNextLine()) {
            saida.println(teclado.nextLine());
        }

        saida.close();
        teclado.close();
        cliente.close();
    }
}
```

```
public class Recebedor implements Runnable {

    private InputStream servidor;

    public Recebedor(InputStream servidor) {
        this.servidor = servidor;
    }

    public void run() {
        // recebe msgs do servidor e imprime na tela
        Scanner s = new Scanner(this.servidor);
        while (s.hasNextLine()) {
            System.out.println(s.nextLine());
        }
    }
}
```

Já o Servidor sofreu bastante modificações. A classe `TrataCliente` é a responsável por cuidar de cada cliente conectado no sistema:

```
public class Servidor {

    public static void main(String[] args) throws IOException {
        // inicia o servidor
        new Servidor(12345).executa();
    }

    private int porta;
    private List<PrintStream> clientes;

    public Servidor (int porta) {
        this.porta = porta;
        this.clientes = new ArrayList<PrintStream>();
    }

    public void executa () throws IOException {
        ServerSocket servidor = new ServerSocket(this.porta);
        System.out.println("Porta 12345 aberta!");

        while (true) {
            // aceita um cliente
            Socket cliente = servidor.accept();
            System.out.println("Nova conexão com o cliente " +
                cliente.getInetAddress().getHostAddress()
            );

            // adiciona saida do cliente à lista
            PrintStream ps = new PrintStream(cliente.getOutputStream());
            this.clientes.add(ps);
```

```java
            // cria tratador de cliente numa nova thread
            TrataCliente tc =
                    new TrataCliente(cliente.getInputStream(), this);
            new Thread(tc).start();
        }

    }

    public void distribuiMensagem(String msg) {
        // envia msg para todo mundo
        for (PrintStream cliente : this.clientes) {
            cliente.println(msg);
        }
    }
}

public class TrataCliente implements Runnable {

    private InputStream cliente;
    private Servidor servidor;

    public TrataCliente(InputStream cliente, Servidor servidor) {
        this.cliente = cliente;
        this.servidor = servidor;
    }

    public void run() {
        // quando chegar uma msg, distribui pra todos
        Scanner s = new Scanner(this.cliente);
        while (s.hasNextLine()) {
            servidor.distribuiMensagem(s.nextLine());
        }
        s.close();
    }
}
```

CAPÍTULO 20

APÊNDICE - PROBLEMAS COM CONCORRÊNCIA

"Quem pouco pensa, engana-se muito." -- Leonardo da Vinci

20.1 THREADS ACESSANDO DADOS COMPARTILHADOS

O uso de Threads começa a ficar interessante e complicado quando precisamos compartilhar objetos entre várias Threads.

Imagine a seguinte situação: temos um Banco com milhões de Contas Bancárias. Clientes sacam e depositam dinheiro continuamente, 24 horas por dia. No primeiro dia de cada mês, o Banco precisa atualizar o saldo de todas as Contas de acordo com uma taxa específica. Para isso, ele utiliza o `AtualizadorDeContas` que vimos anteriormente.

O `AtualizadorDeContas`, basicamente, pega uma a uma cada uma das milhões de contas e chama seu método `atualiza`. A atualização de milhões de contas é um processo demorado, que dura horas; é inviável parar o banco por tanto tempo até que as atualizações tenham completado. É preciso executar as atualizações paralelamente às atividades, de depósitos e saques, normais do banco.

Ou seja, teremos várias threads rodando paralelamente. Em uma thread, pegamos todas as contas e vamos chamando o método `atualiza` de cada uma. Em outra, podemos estar sacando ou depositando dinheiro. Estamos compartilhando objetos entre múltiplas threads (as contas, no nosso caso).

Imagine a seguinte possibilidade (mesmo que muito remota): no exato instante em que o atualizador está atualizando uma Conta X, o cliente dono desta Conta resolve efetuar um saque. Como sabemos, ao trabalhar com Threads, o escalonador pode parar uma certa Thread a qualquer instante para executar outra, e você não tem controle sobre isso.

Veja essa classe Conta:

```
public class Conta {

    private double saldo;

        // outros métodos e atributos...

        public void atualiza(double taxa) {
```

```
            double saldoAtualizado = this.saldo * (1 + taxa);
            this.saldo = saldoAtualizado;
        }

        public void deposita(double valor) {
            double novoSaldo = this.saldo + valor;
            this.saldo = novoSaldo;
        }
}
```

Imagine uma Conta com saldo de 100 reais. Um cliente entra na agência e faz um depósito de 1000 reais. Isso dispara uma Thread no banco que chama o método `deposita()` ; ele começa calculando o `novoSaldo` que passa a ser 1100 (linha 13). Só que por algum motivo que desconhecemos, o escalonador pára essa thread.

Neste exato instante, ele começa a executar uma outra Thread que chama o método `atualiza` da mesma `Conta` , por exemplo, com taxa de 1%. Isso quer dizer que o `novoSaldo` passa a valer 101 reais (linha 8). E, nesse instante o escalonador troca de Threads novamente. Ele executa a linha 14 na Thread que fazia o depósito; o saldo passa a valer 1100. Acabando o deposita, o escalonador volta pra Thread do atualiza e executa a linha 9, fazendo o saldo valer 101 reais.

Resultado: o depósito de mil reais foi totalmente ignorado e seu Cliente ficará pouco feliz com isso. Perceba que não é possível detectar esse erro, já que todo o código foi executado perfeitamente, sem problemas. **O problema, aqui, foi o acesso simultâneo de duas Threads ao mesmo objeto**.

E o erro só ocorreu porque o escalonador parou nossas Threads naqueles exatos lugares. Pode ser que nosso código fique rodando 1 ano sem dar problema algum e em um belo dia o escalonador resolve alternar nossas Threads daquela forma. Não sabemos como o escalonador se comporta! Temos que proteger nosso código contra esse tipo de problema. Dizemos que essa classe não é *thread safe*, isso é, não está pronta para ter uma instância utilizada entre várias threads concorrentemente.

O que queríamos era que não fosse possível alguém atualizar a `Conta` enquanto outra pessoa está depositando um dinheiro. Queríamos que uma Thread não pudesse mexer em uma `Conta` enquanto outra Thread está mexendo nela. Não há como impedir o escalonador de fazer tal escolha. Então, o que fazer?

20.2 CONTROLANDO O ACESSO CONCORRENTE

Uma ideia seria criar uma **trava** e, no momento em que uma Thread entrasse em um desses métodos, ela trancaria a entrada com uma chave. Dessa maneira, mesmo que sendo colocada de lado, nenhuma outra Thread poderia entrar nesses métodos, pois a chave estaria com a outra Thread.

Essa ideia é chamada de **região crítica**. É um pedaço de código que definimos como crítico e que não pode ser executado por duas threads ao mesmo tempo. Apenas uma thread por vez consegue entrar em alguma região crítica.

Podemos fazer isso em Java. Podemos usar qualquer objeto como um **lock** (trava, chave), para poder **sincronizar** em cima desse objeto, isto é, se uma Thread entrar em um bloco que foi definido como sincronizado por esse lock, apenas uma Thread poderá estar lá dentro ao mesmo tempo, pois a chave estará com ela.

A palavra chave `synchronized` dá essa característica a um bloco de código e recebe qual é o objeto que será usado como chave. A chave só é devolvida no momento em que a Thread que tinha essa chave sair do bloco, seja por `return` ou disparo de uma exceção (ou ainda na utilização do método `wait()`)..

Queremos, então, bloquear o acesso simultâneo a uma mesma `Conta`:

```
public class Conta {

    private double saldo;

    // outros métodos e atributos...

    public void atualiza(double taxa) {
        synchronized (this) {
            double saldoAtualizado = this.saldo * (1 + taxa);
            this.saldo = saldoAtualizado;
```

```
        }
    }

    public void deposita(double valor) {
        synchronized (this) {
            double novoSaldo = this.saldo + valor;
            this.saldo = novoSaldo;
        }
    }
}
```

Observe o uso dos blocos `synchronized` dentro dos dois métodos. Eles bloqueiam uma Thread utilizando o mesmo objeto `Conta` , o `this` .

Esses métodos são mutuamente exclusivos e só executam de maneira atômica. Threads que tentam pegar um lock que já está pego, ficarão em um conjunto especial esperando pela liberação do lock (não necessariamente numa fila).

SINCRONIZANDO O BLOCO INTEIRO

É comum sempre sincronizarmos um método inteiro, normalmente utilizando o `this` .

```
public void metodo() {
    synchronized (this) {
        // conteúdo do metodo
    }
}
```

Para este mesmo efeito, existe uma sintaxe mais simples, onde o `synchronized` pode ser usado como modificador do método:

```
public synchronized void metodo() {
    // conteúdo do metodo
}
```

MAIS SOBRE LOCKS, MONITORES E CONCORRÊNCIA

Se o método for estático, será sincronizado usando o lock do objeto que representa a classe (`NomeDaClasse.class`).

Além disso, o pacote `java.util.concurrent` , conhecido como **JUC**, entrou no Java 5.0 para facilitar uma série de trabalhos comuns que costumam aparecer em uma aplicação concorrente.

Esse pacote ajuda até mesmo criar threads e pool de threads, através dos Executors.

20.3 VECTOR E HASHTABLE

Duas collections muito famosas são `Vector` e `Hashtable`, a diferença delas com suas irmãs `ArrayList` e `HashMap` é que as primeiras são thread safe.

Você pode se perguntar porque não usamos sempre essas classes thread safe. Adquirir um lock tem um custo, e caso um objeto não vá ser usado entre diferentes threads, não há porque usar essas classes que consomem mais recursos. Mas nem sempre é fácil enxergar se devemos sincronizar um bloco, ou se devemos utilizar blocos sincronizados.

Antigamente o custo de se usar locks era altíssimo, hoje em dia isso custa pouco para a JVM, mas não é motivo para você sincronizar tudo sem necessidade.

20.4 UM POUCO MAIS...

- Você pode mudar a prioridade de cada uma de suas Threads, mas isto também é apenas uma sugestão ao escalonador.
- Existe um método `stop` nas Threads, porque não é boa prática chamá-lo?
- Um tópico mais avançado é a utilização de `wait`, `notifiy` e `notifyAll` para que as Threads comuniquem-se de eventos ocorridos, indicando que podem ou não podem avançar de acordo com condições
- O pacote `java.util.concurrent` foi adicionado no Java 5 para facilitar o trabalho na programação concorrente. Ele possui uma série de primitivas para que você não tenha de trabalhar diretamente com `wait` e `notify`, além de ter diversas coleções thread safe.

20.5 EXERCÍCIOS AVANÇADOS DE PROGRAMAÇÃO CONCORRENTE

Exercícios só recomendados se você já tinha algum conhecimento prévio de programação concorrente, locks, etc.

1. Vamos enxergar o problema ao se usar uma classe que não é *thread-safe*: a `ArrayList` por exemplo.

   Imagine que temos um objeto que guarda todas as mensagens que uma aplicação de chat recebeu. Vamos usar uma `ArrayList<String>` para armazená-las. Nossa aplicação é *multi-thread*, então diferentes threads vão inserir diferentes mensagens para serem registradas. Não importa a ordem que elas sejam guardadas, desde que elas um dia sejam!

   Vamos usar a seguinte classe para adicionar as queries:

```
public class ProduzMensagens implements Runnable {
    private int comeco;
    private int fim;
    private Collection<String> mensagens;

    public ProduzMensagens(int comeco, int fim, Collection<String> mensagens) {
        this.comeco = comeco;
        this.fim = fim;
        this.mensagens = mensagens;
    }

    public void run() {
        for (int i = comeco; i < fim; i++) {
            mensagens.add("Mensagem " + i);
        }
    }
}
```

   Vamos criar três threads que rodem esse código, todas adicionando as mensagens na **mesma** `ArrayList`. Em outras palavras, teremos threads compartilhando e acessando um mesmo objeto: é aqui que mora o perigo.

```
public class RegistroDeMensagens {

    public static void main(String[] args) throws InterruptedException {
        Collection<String> mensagens = new ArrayList<String>();

        Thread t1 = new Thread(new ProduzMensagens(0, 10000, mensagens));
        Thread t2 = new Thread(new ProduzMensagens(10000, 20000, mensagens));
        Thread t3 = new Thread(new ProduzMensagens(20000, 30000, mensagens));

        t1.start();
        t2.start();
        t3.start();

        // faz com que a thread que roda o main aguarde o fim dessas
        t1.join();
        t2.join();
        t3.join();
```

```
        System.out.println("Threads produtoras de mensagens finalizadas!");

        // verifica se todas as mensagens foram guardadas
        for (int i = 0; i < 15000; i++) {
            if (!mensagens.contains("Mensagem " + i)) {
                throw new IllegalStateException("não encontrei a mensagem: " + i);
            }
        }

        // verifica se alguma mensagem ficou nula
        if (mensagens.contains(null)) {
            throw new IllegalStateException("não devia ter null aqui dentro!");
        }

        System.out.println("Fim da execucao com sucesso");
    }
}
```

Rode algumas vezes. O que acontece?

2. Teste o código anterior, mas usando `synchronized` ao adicionar na coleção:

```
public void run() {
    for (int i = comeco; i < fim; i++) {
        synchronized (mensagens) {
            mensagens.add("Mensagem " + i);
        }
    }
}
```

3. Sem usar o `synchronized` teste com a classe `Vector`, que **é uma** `Collection` e é *thread-safe*.

   O que mudou? Olhe o código do método `add` na classe `Vector`. O que tem de diferente nele?

4. Novamente sem usar o `synchronized`, teste usar `HashSet` e `LinkedList`, em vez de `Vector`. Faça vários testes, pois as threads vão se entrelaçar cada vez de uma maneira diferente, podendo ou não ter um efeito inesperado.

   No capítulo de Sockets usaremos threads para solucionar um problema real de execuções paralelas.

CAPÍTULO 21

APÊNDICE - INSTALAÇÃO DO JAVA

"Quem pouco pensa, engana-se muito." -- Leonardo da Vinci

Como vimos antes, a VM é apenas uma especificação e devemos baixar uma implementação. Há muitas empresas que implementam uma VM, como a própria Oracle, a IBM, a Apache e outros.

A da Oracle é a mais usada e possui versões para Windows, Linux e Solaris. Você pode baixar o SDK acessando:

http://www.oracle.com/technetwork/java/

Nesta página da Oracle, você deve escolher o Java SE, dentro dos top downloads. Depois, escolha o JDK e seu sistema operacional.

21.1 INSTALANDO NO UBUNTU E EM OUTROS LINUX

Cada distribuição Linux tem sua própria forma de instalação. Algumas já trazem o Java junto, outras possibilitam que você instale pelos repositórios oficiais e em alguns casos você precisa baixar direto da Oracle e configurar tudo manualmente.

No Ubuntu, a distribuição usada na Caelum, a instalação é bastante simples. Basta ir no terminal e digitar:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

Caso prefira utilizar o openjdk, a distribuição opensource, basta fazer

```
sudo apt-get install openjdk-7-jdk
```

Por enquanto ele possui apenas a versão 7. No linux fedora, você faria com `su -c "yum install java-1.7.0-openjdk"` .

Se você já tiver outras versões instaladas no seu Ubuntu, pode utilizar `sudo update-alternatives --config java` para escolher entre elas.

Uma instalação mais braçal, sem usar repositório , pode ser feita baixando o instalador no próprio site da Oracle. É um `tar.gz` que possui um `.bin` que deve ser executado. Depois, é necessário

apontar `JAVA_HOME` para esse diretório e adicionar `JAVA_HOME/bin` no seu `PATH` .

21.2 NO MAC OS X

O Mac OS X já traz o Java instalado junto com o sistema operacional até a versão 10.6. As versões mais novas, do Lion em diante, o instalador do Mac vai perguntar se você deseja baixá-lo quando for rodar sua primeira aplicação Java, como o Eclipse.

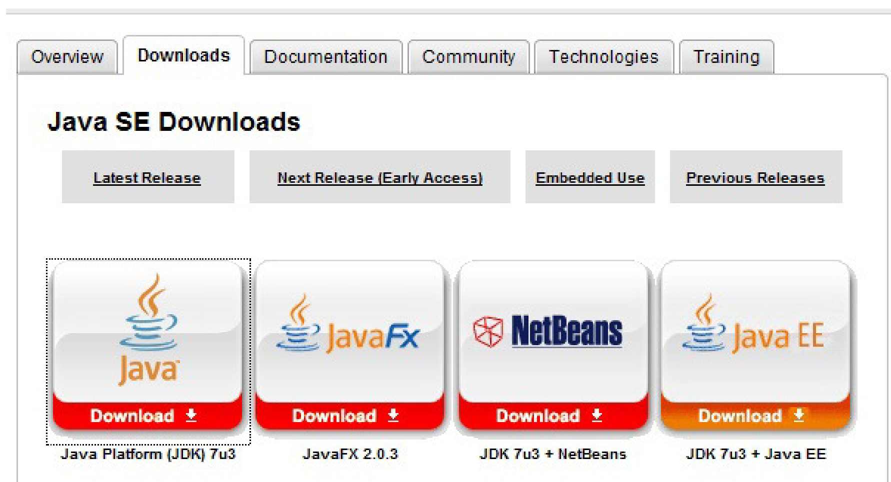
A versão para o Java 8 pode ser baixada no mesmo site:

http://www.oracle.com/technetwork/java/

21.3 INSTALAÇÃO DO JDK EM AMBIENTE WINDOWS

Para instalar o JDK no Windows, primeiro baixe-o no site da Oracle. É um simples arquivo executável que contém o Wizard de instalação:

http://www.oracle.com/technetwork/java/



Instalação

- Dê um clique duplo no arquivo `jdk-<versão>-windows-i586-p.exe` e espere até ele entrar no wizard de instalação.



- Aceite os próximos dois passos clicando em *Next*. Após um tempo, o instalador pedirá para escolher em que diretório instalar o SDK. Pode ser onde ele já oferece como padrão. Anote qual foi o diretório escolhido, vamos utilizar esse caminho mais adiante. A cópia de arquivos iniciará:



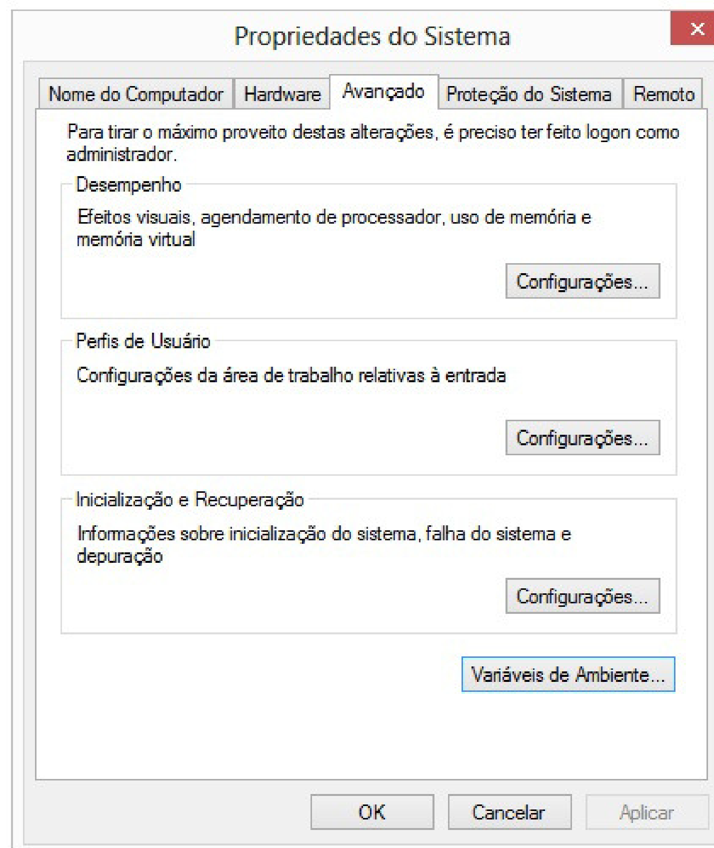
- O instalador instalará também o JavaFX 2. Após isso, você será direcionado à uma página onde você pode, opcionalmente, criar uma conta na Oracle para registrar sua instalação.

Configurando o ambiente

Precisamos configurar algumas variáveis de ambiente após a instalação, para que o compilador seja acessível via linha de comando. Caso você vá utilizar diretamente o Eclipse, provavelmente não será necessário realizar esses passos.

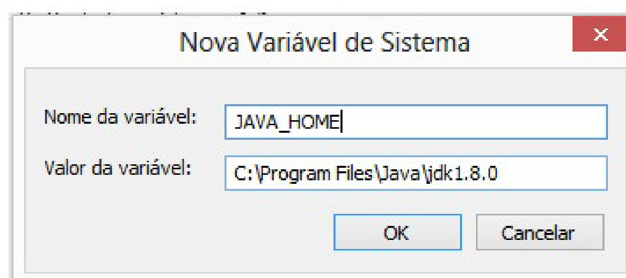
- Clique com o botão direito em cima do ícone *Computador* e selecione a opção *Propriedades*.

- Escolha a aba "Configurações Avançadas de Sistema" e depois clique no botão "Variáveis de Ambiente"



- Nesta tela, você verá, na parte de cima, as variáveis de ambiente do usuário corrente e, embaixo, as variáveis de ambiente do computador (servem para todos os usuários). Clique no botão *Novo...* da parte de baixo.

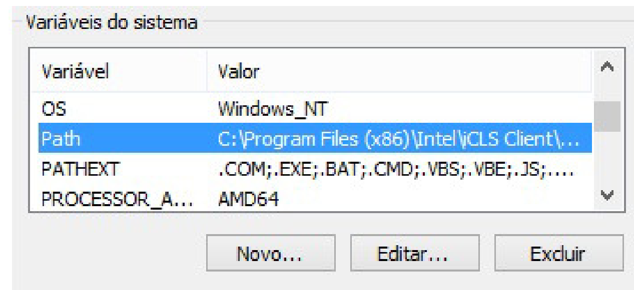
- Em *Nome da Variável* digite `JAVA_HOME` e, em valor da variável, digite o caminho que você utilizou na instalação do Java. Provavelmente será algo como: `C:\Program Files\Java\jdk1.8.0_03` :



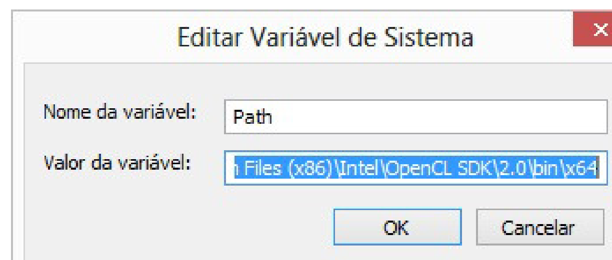
Clique em *Ok*.

- Não vamos criar outra variável, mas sim *alterar*. Para isso, procure a variável PATH, ou Path (dá

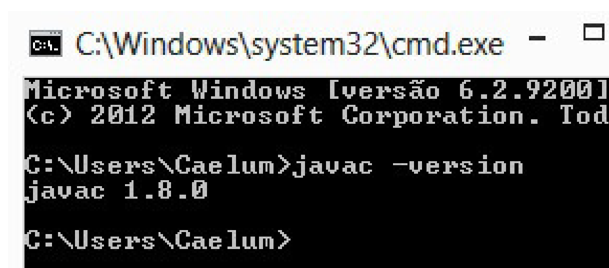
no mesmo), e clique no botão de baixo "Editar".



- Não altere o nome da variável! Deixe como está e adicione no final do valor `;%JAVA_HOME%\bin` , não esqueça do ponto-e-vírgula - assim, você está adicionando mais um caminho à sua variável Path.



- Abra o prompt, indo em *Iniciar*, *Executar* e digite `cmd` .
- No console, digite `javac -version` . O comando deve mostrar a versão do Java Compiler e algumas opções.



Você pode seguir para a instalação do Eclipse, conforme visto no seu capítulo, ou utilizar um editor de texto simples como o bloco de notas para os primeiros capítulos de apostila.

Qualquer dúvida, não hesite de postá-la no Grupo de Usuários Java:

http://www.guj.com.br.

Capítulo 22

APÊNDICE - DEBUGGING

"Olho por olho, e o mundo acabará cego."--Mohandas Gandhi

22.1 O QUE É DEBUGAR

Debugging (em português, depuração ou depurar) é um processo de reduzir ou encontrar bugs no seu sistema. De uma forma geral, debugging não é uma tarefa fácil de ser executada. Muitas variações podem atrapalhar esse processo, por exemplo, a linguagem que estamos utilizando e ferramentas disponíveis para fazermos debugging de um código.

O Java em si facilita muito neste processo, pois nos fornece maneiras de sabermos se o código está errado, por exemplo as exceptions. Em linguagens de baixo nível saber onde o bug estava era extremamente complicado. O que também facilita nosso trabalho são as ferramentas de debug. Veremos que elas são necessárias nos casos que nossos testes de unidade de logging não foram suficientes para encontrar a razão de um problema.

22.2 DEBUGANDO NO ECLIPSE

No curso utilizamos o Eclipse como IDE para desenvolvermos nosso código. Como foi dito ferramentas de debugging facilitam muito nosso trabalho, o Eclipse é uma das IDEs mais poderosas do mercado e nos fornece uma ferramenta que torna o processo extremamente simples.

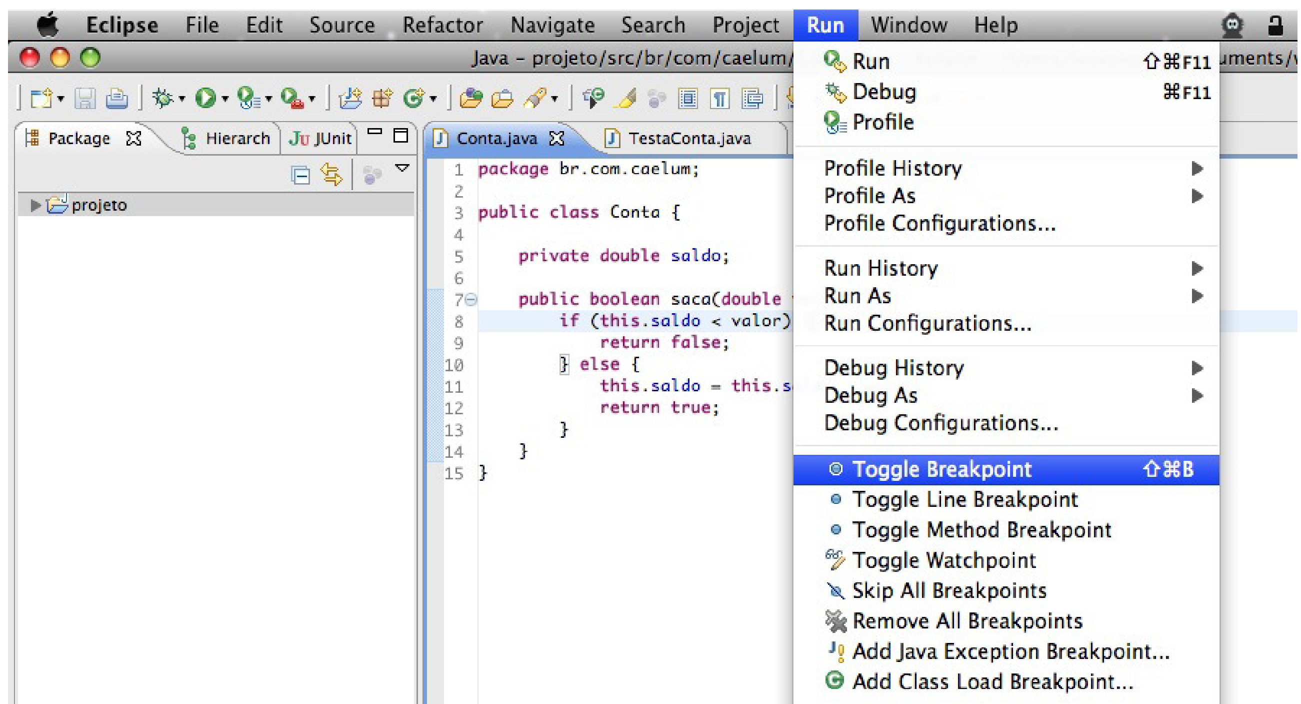
O primeiro recurso que temos que conhecer quando começamos a debugar no Eclipse são os **breakpoints**. Eles são pontos de partida em nosso código para iniciarmos o processo de debug. Por exemplo, no código abaixo, imagine que desejamos debugar o comportamento do método `saca` da classe `Conta` , mais especificamente do `if` que verifica se saldo é menor que o valor a ser sacado. Colocaríamos o **breakpoint** exatamente na linha `if (this.saldo < valor) {` :

```
public class Conta {

    private double saldo;

    public boolean saca(double valor) {
        if (this.saldo < valor) {
            return false;
        } else {
            this.saldo = this.saldo - valor;
            return true;
        }
    }
}
```

Mas como faço isso? Muito simples, basta clicar na linha que deseja adicionar o breakpoint, depois clicar no menu **Run -> Toogle Breakpoint**.



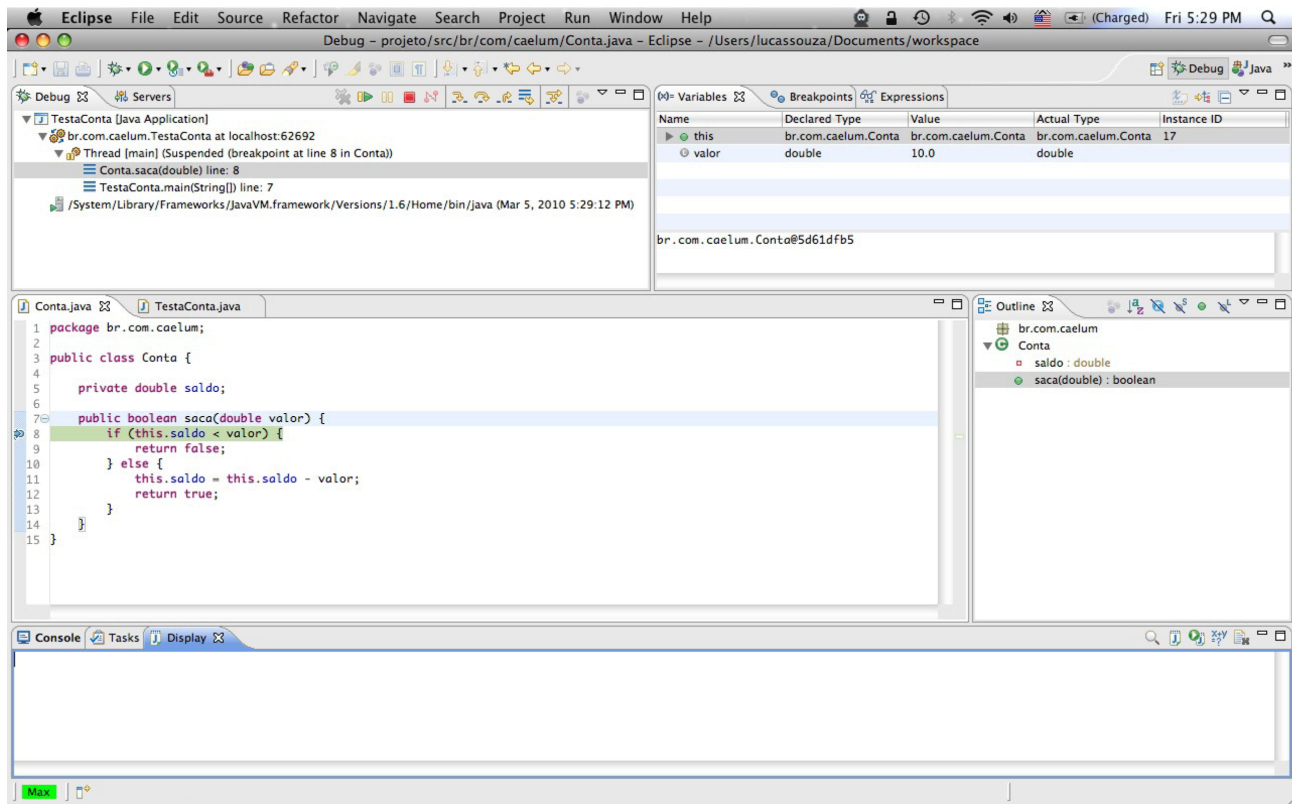
Esse é o tipo mais clássico de breakpoint, veremos alguns outros ao longo do capítulo.

Agora que já adicionamos o breakpoint que é o ponto de partida, vamos debugar nosso código. Precisamos rodar nosso código, ou seja, chamar o método `saca` para que o breakpoint seja encontrado. Teremos um código similar ao seguinte:

```
public class TestaConta {
```

```
    public static void main(String[] args) {
        Conta conta = new Conta();
        conta.saca(200);
    }
}
```

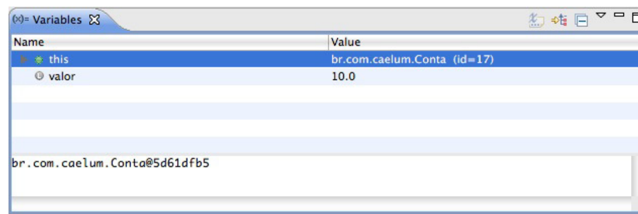
O processo normal para executarmos esse código seria clicar no menu *Run -> Run As -> Java Application*. Porém para rodar o nosso código em **modo debug** e ativar nosso breakpoint, devemos rodar o codigo no menu *Run -> Debug As -> Java Application*. Quando um breakpoint for encontrado no código que está sendo executado, o eclipse exibirá uma perspectiva específica de debug, apontando para a linha que tem o breakpoint.



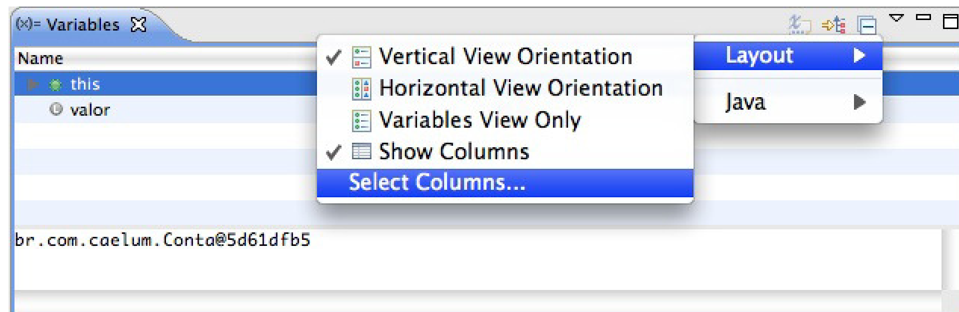
22.3 PERSPECTIVA DE DEBUG

Temos várias informações disponíveis nessa perspectiva, algumas são essenciais e básicas para trabalharmos com debug no nosso dia-a-dia, outras não tão relevantes e só usamos em casos muito específicos.

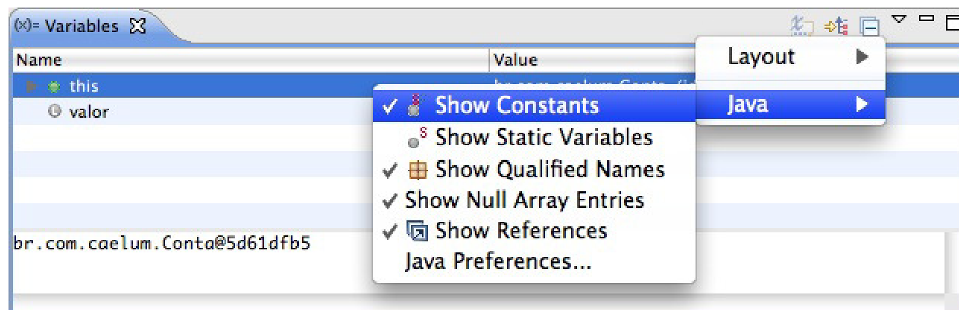
Dentro da perspectiva de debug, temos uma aba chamada `Variables`. São exibidas todas as variáveis encontradas dentro do código que você está debugando. Por exemplo, no debug que fizemos serão exibidas as variáveis do método `saca`, neste caso, `valor`. Além dos atributos de instância do objeto.



Podemos exibir mais informações sobre as variáveis, basta adicionarmos as colunas que desejamos na tabela exibida.



É possível também adicionarmos constantes e variáveis estáticas da classe que está sendo debugada.

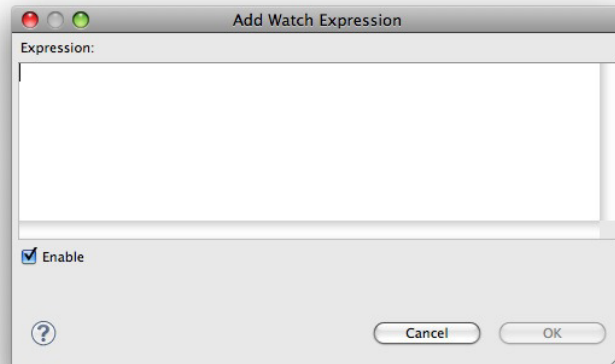


Na aba `Breakpoints` são exibidos todos os breakpoints que seu workspace possui. Mas por que isso é importante? É importante porque podemos ver todos os pontos de debug presentes e melhor, podemos desabilitá-los um a um ou todos de uma só vez. Você pode até mesmo pedir para exportar os breakpoints.

Para desabilitar ou habilitar todos breakpoints basta clicarmos no ícone **Skip All Breakpoints**. Se quisermos desabilitar um a um, basta desmarcar o checkbox e o breakpoint será desativado. Às vezes, encontrar o código onde o breakpoint foi colocado pode ser complicado, na aba `Breakpoints` isso fica bem fácil de fazer, basta dar um duplo clique no breakpoint e o eclipse automaticamente nos mostra a classe "dona" dele.

Quando estamos debugando código, muitas vezes é interessante saber o valor de alguma expressão ou método. Por exemplo, uma condição dentro de um if, `this.saldo > valor` . Esse valor não está em

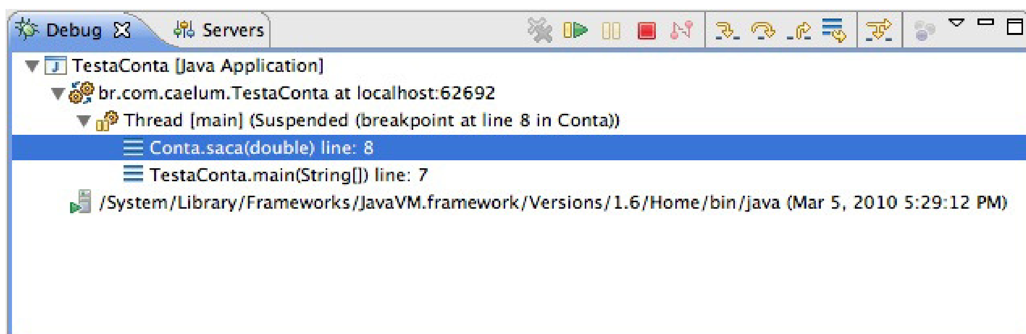
uma variável, ele está em uma expressão, o que pode tornar saber o valor dela complicado. A feature de `Expressions` descomplica esse processo para nós. Na perspectiva de Debug temos a aba `Expressions`. Basta clicar com o direito dentro da aba, e clicar em **Add Expression**:



E o resultado da expressão é exibido.

Temos outra aba importante chamada de `Debug`. Dentre as funções dela estão:

- **Threads** - Exibe as threads que estão sendo executadas, e melhor, mostra qual thread efetuou a chamada para o método onde está o debug. Além disso mostra a pilha de execução, o que nos permite voltar a chamada de um método
- **Barra de navegação** - Que permite alterarmos os caminhos que o debug seguirá.



A lista a seguir mostrar algumas teclas e botões que alteram o caminho natural dos nosss debug:

- **F5** - Vai para o próximo passo do seu programa. Se o próximo passo for um método, ele entrará no código associado;
- **F6** - Também vai para o próximo passo, porém se o próximo passo for um método, ele não entrará no código associado;
- **F7** - Voltará e mostrará o método que fez a chamada para o código que está sendo debugado. No nosso caso voltará para o método `main` da classe `TestaConta`;

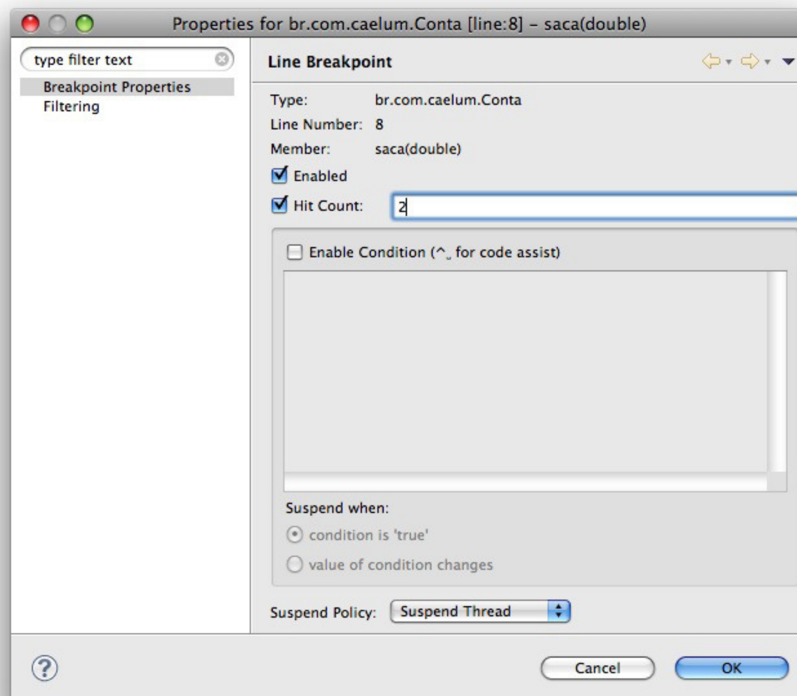
- **F8** - Vai para o próximo breakpoint, se nenhum for encontrado, o programa seguirá seu fluxo de execução normal.

Você também pode usar os botões que estão presentes na aba `Debug` .



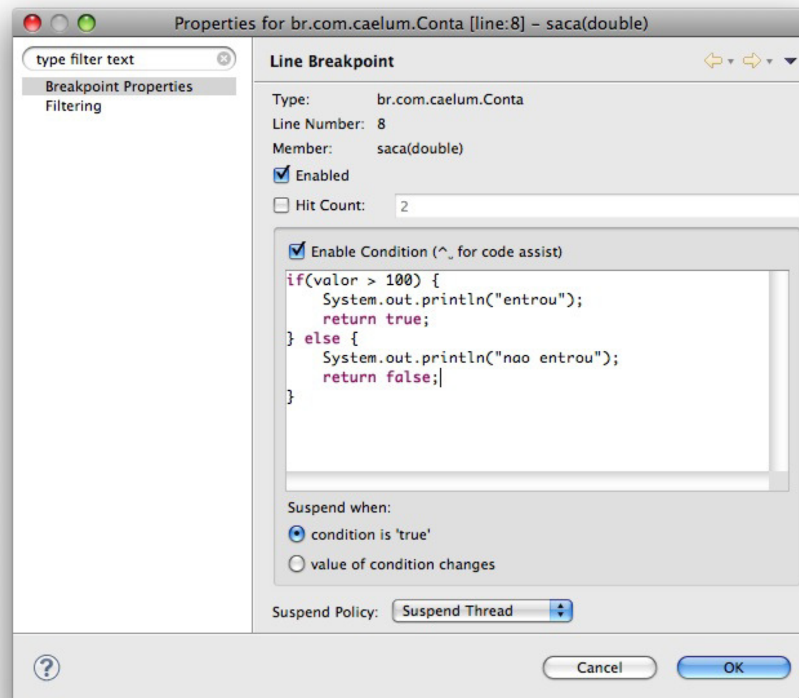
22.4 DEBUG AVANÇADO

Depois que colocamos um breakpoint em algum ponto do nosso código, podemos colocar algumas propriedades nele, por exemplo, usar alguma condição para restringir quando o breakpoint será ativado em tempo de execução. Podemos restringir na propriedade **Hit Count** que o breakpoint só será ativado quando a linha em que ele encontra-se for executada 'X' vezes.



Como na imagem acima o breakpoint só será ativado quando a linha de código em que ele se encontra for executada '2' vezes. Podemos também colocar alguma expressão condicional, um `if` , por

exemplo.



O breakpoint, neste caso, somente será ativado quando o argumento `valor` que foi passado ao método `saca` for maior que 100. O importante aqui é notarmos que devemos retornar **sempre** um valor booleano, se não o fizermos, teremos um erro em tempo de execução. Essa propriedade é válida quando queremos colocar aqueles famosos `System.out.println("entrou no if tal")` para efeito de log, podemos fazer isso colocando o log dentro da expressão condicional nas propriedades do breakpoint.

O display é uma das partes mais interessantes do debug do eclipse, ele provê uma maneira de executarmos qualquer código que quisermos quando estamos em debugging. Criar uma classe, instanciar objetos dessa classe, utilizar if's, for's, while's, todos os recursos do Java, além de poder utilizar as variáveis, métodos, constantes da classe que estamos debugando.

Um exemplo clássico é quando estamos em debugging e queremos saber o retorno de algum método do qual não temos acesso, o que faríamos antes seria colocar um amontoado de `System.out.println` , poluindo extremamente nosso código. No display o que fazemos é efetuar a chamada desse código e automaticamente os resultados são exibidos.

Para vermos um efeito real disso, vamos alterar um pouco o comportamento da classe Conta, de modo que agora o saldo para saque tenha que ser o saldo real mais o valor do limite. Nosso código fica assim:

```
public class Conta {
```

```
    private double saldoReal;
    private double limite;

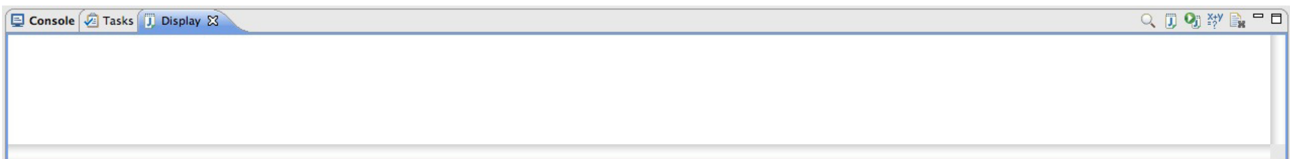
    public Conta(double limite) {
        this.limite = limite;
    }

    public boolean saca(double valor) {
        if (!isSaldoSuficiente(valor)) {
            return false;
        } else {
            this.saldoReal = this.saldoReal - valor;
            return true;
        }
    }

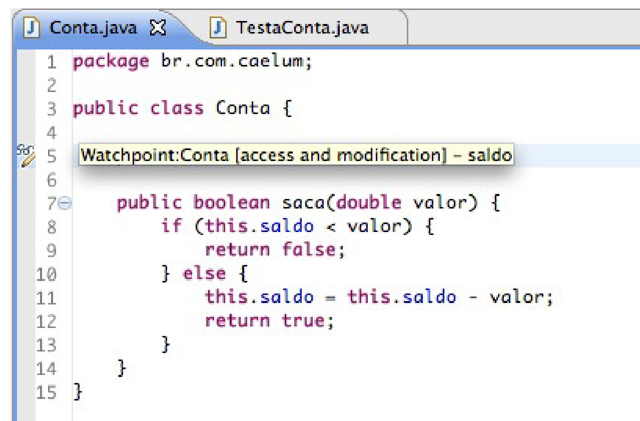
    private boolean isSaldoSuficiente(double valor) {
        return (this.saldoReal + this.limite) > valor;
    }
}
```

Repare que o `if` que verifica se o saldo é suficiente para efetuarmos o saque chama um método `isSaldoSuficiente` , o que pode ser um problema quando estamos debugando, afinal a condição do `if` é um método. Se utilizarmos o display podemos fazer a chamada do método `isSaldoSuficiente` , ver seu resultado e o melhor, não afetamos o debug, apenas queremos ver o resultado do método, por exemplo.

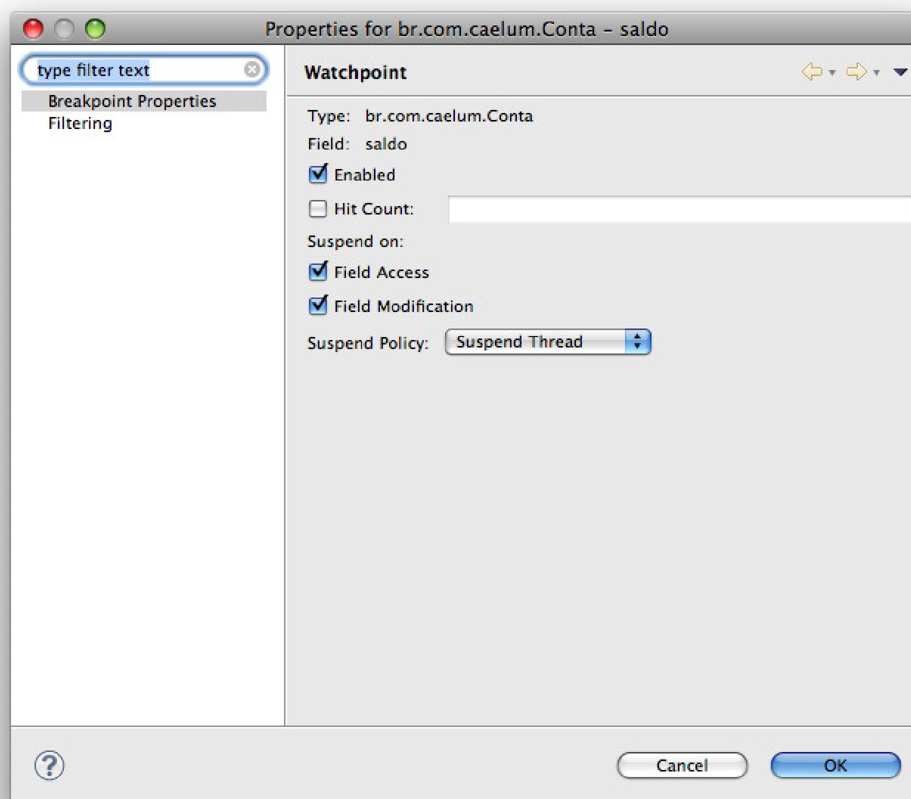
Para exibirmos a aba **Display** é bem simples. Tecle **Ctrl + 3**, digite **Display** e a aba será exibida. Quando rodarmos nosso código em modo debug, podemos ir no display, digitarmos uma chamada para o método `isSaldoSuficiente` , executamos esse código que foi digitado selecionando-o dentro do display e teclando **Ctrl + Shift + D** e o resultado será impresso, assim como na imagem abaixo:



Muitas vezes queremos "seguir" alguma variável de instância, ou seja, qualquer chamada para essa variável (leitura ou escrita) queremos ser notificados disso. Podemos usar o watchpoint, que fará nosso programa entrar em modo debug, quando qualquer alteração na variável que estamos seguindo ocorrer, o programa entrará em debug exatamente na linha que fez a alteração. Para colocarmos um watchpoint, basta dar um duplo clique no atributo de instância que deseja colocá-lo.



É possível alterar esse comportamento padrão, e definir se você quer que o watchpoint seja ativado para leitura ou somente para escrita.



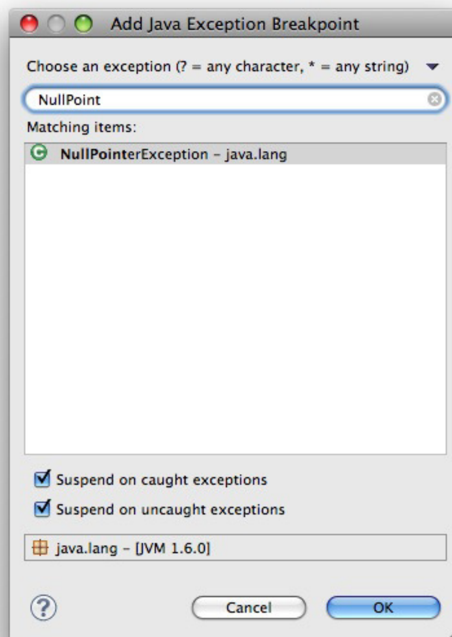
A idéia desse tipo de breakpoint é fazer nosso programa entrar em debug quando alguma exceção específica ocorrer. Quando definirmos essa exceção no **Exception Breakpoint** e a mesma ocorrer, automaticamente nosso programa entra em debug na linha que gerou aquela exceção. Por exemplo, vamos alterar o codigo da classe `TestaConta` para que a mesma tenha uma `NullPointerException` :

```
public class TestaConta {
    public static void main(String[] args) {
        Conta conta = null;
        conta.saca(10);
    }
}
```

Quando rodarmos o código acima, teremos uma `NullPointerException` . Pode ser útil nesses casos debugar e saber onde a exceção está ocorrendo de fato, em qual linha mais especificamente. Para fazermos isso podemos criar um Exception Breakpoint, que debugará códigos que eventualmente lancem uma `NullPointerException` , por exemplo. Basta abrirmos a aba **Breakpoints** e clicarmos no ícone abaixo:

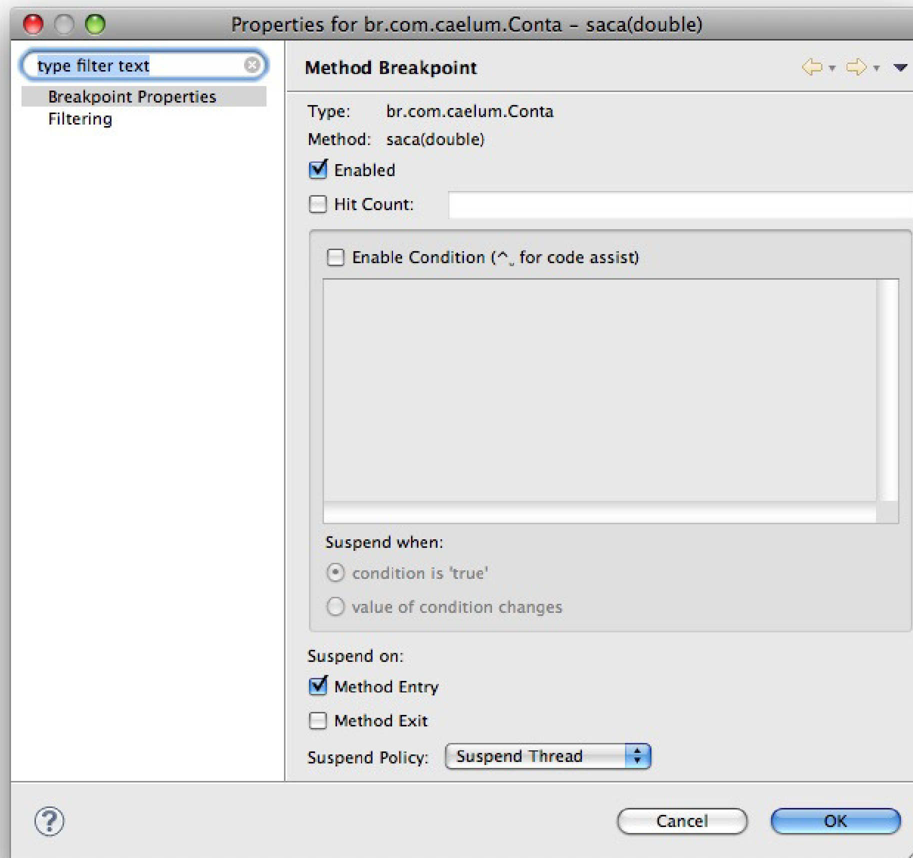


Será aberta uma janela onde podemos buscar por uma exceção específica.

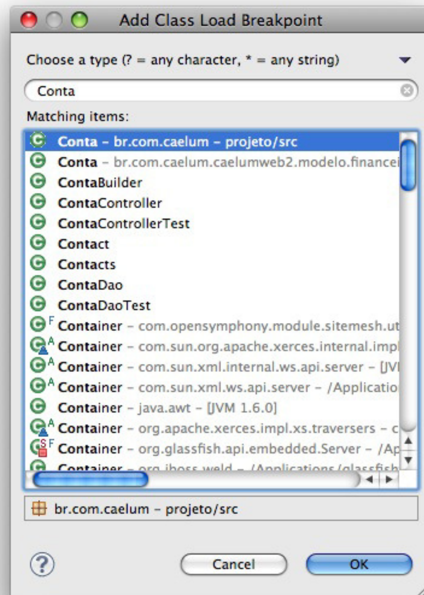


Podemos definir um breakpoint que é ativado ou antes ou depois que o método é chamado. Para definirmos ele, basta estar em qualquer parte do método que desejamos debugar, clicar no menu **Run ->**

Toogle Method Breakpoint. Podemos editar as propriedades desse breakpoint dizendo se queremos que ele seja ativado antes(default) ou depois da execução do método. Basta acessar as propriedades do method breakpoint e alterá-las.



É util quando desejamos que um breakpoint seja ativado quando uma classe específica for carregada pela primeira vez, chamamos esse breakpoint de **Class Breakpoint**. Basta clicarmos no menu **Run -> Add Class Load Breakpoint**, uma janela será aberta e basta digitarmos o nome da classe e adicionarmos:



22.5 PROFILING

Um dos principais hábitos que nós desenvolvedores devemos evitar é a questão da otimização prematura, ou seja, quando desenvolvemos uma aplicação para um cliente, devemos nos preocupar em **atender o requisitos funcionais de maneira mais rápida e mais simples possível**. O passo seguinte é refatorar seu código para que ele seja melhorado e para que no futuro possa se adaptar as possíveis mudanças.

A regra é: "Deixe os problemas do futuro, para serem resolvidos no futuro".

Uma das ferramentas que nos auxiliam na questão de não otimizar nosso código prematuramente, são as ferramentas de profiling, que tornam aparentes, por exemplo, os problemas de memória e cpu, que podem fazer com que otimizemos nosso código. Atualmente devido as técnicas que utilizamos para entregar algo de valor para o cliente, focamos principalmente na qualidade, aspectos funcionais, testes, etc. Porém, muitos problemas que não fazem parte dos requisitos funcionais podem acontecer apenas quando a aplicação está em produção, neste ponto as ferramentas de profiling também nos ajudam.

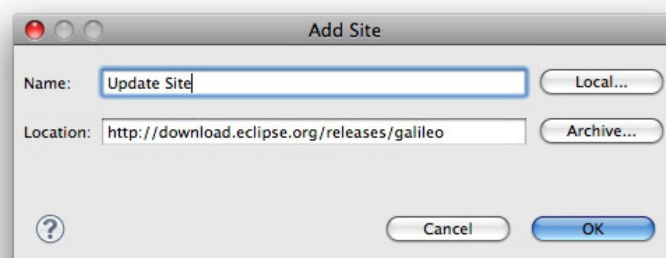
22.6 PROFILING NO ECLIPSE TPTP

Juntamente com o Eclipse temos a opção de instalar e utilizar uma ferramenta de profiling conhecida como Eclipse TPTP (Eclipse Test & Performance Tools Platform), que nos fornece opções para isolar e identificar problemas de performance, tais como: memória (memory leak), recursos e processamento. O TPTP nos permite analisar de simples aplicações java até aplicações que rodam em múltiplas máquinas e em múltiplas plataformas.

Alternativas ao TPTP

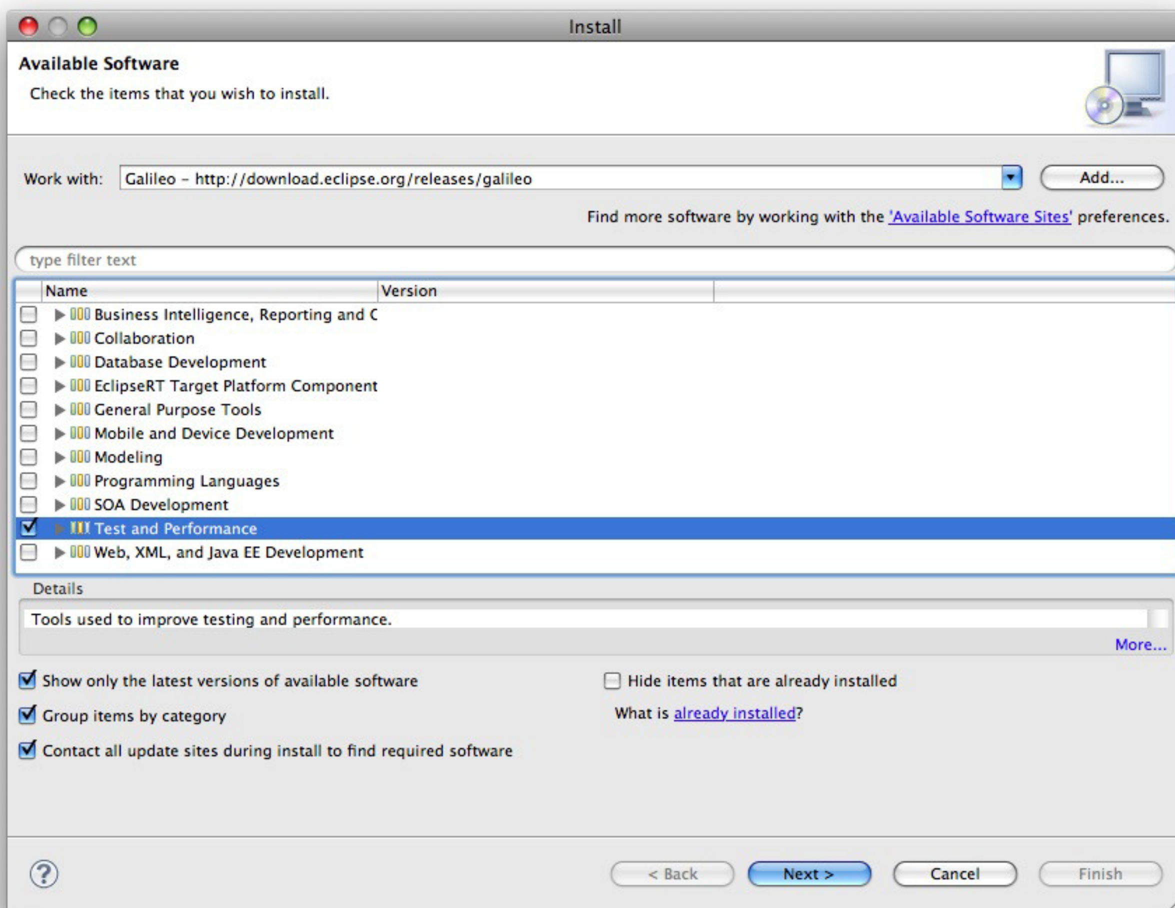
Existem algumas alternativas ao TPTP, os mais conhecidos são Netbeans Profiler (http://profiler.netbeans.org/) que é gratuito, e o JProfiler (http://www.ej-technologies.com/products/jprofiler/overview.html) que é pago.

O TPTP não vem por padrão junto com o Eclipse. Portanto, para utilizarmos é necessário a instalação do mesmo. Podemos fazer o processo de instalação de duas maneiras. A primeira e mais fácil é utilizando o Update Site do Eclipse que resolve as possíveis dependências e nos possibilita escolher quais features queremos instalar. Para instalar o TPTP através desse recurso, basta ir no menu: *Help -> Install New Software*, uma janela será aberta, basta clicar em Add... e preenchê-la conforme a imagem a seguir:



Basta adicionar as ferramentas do TPTP em nosso eclipse, para isto, selecione o repositório que

acabamos de adicionar e a versão do TPTP que queremos instalar, neste caso, a versão 4.6.2.



Instalando pelo Zip

Você tem a opção de instalar o TPTP baixando o zip do projeto e colocando manualmente no diretório de instalação do seu eclipse. Mais informações no link: http://www.eclipse.org/tptp/home/downloads/4.6.0/documents/installguide/InstallGuide46.html

Um problema que pode acontecer em aplicações e que muitas pessoas não conhecem a fundo, é a questão do pool de Strings que pode eventualmente ficar muito grande. Este problema pode ser causado porque objetos do tipo String são imutáveis, sendo assim, se fizermos concatenações de Strings muitas vezes, cada uma dessas concatenações produzirá uma nova String, que automaticamente será colocada no pool da JVM.

A alternativa neste caso, seria trabalhar com objetos do tipo StringBuilder ou StringBuffer que

funcionam como Strings, mas que não produzem Strings novas em caso de uma concatenação. Mas como medir o tamanho do nosso pool de String?

O TPTP possui uma aba de estatísticas que nos mostra o tempo que um método levou para ser executado, quanto processamento esse método gastou, quanto de memória foi gasto com cada método. Vamos analisar algumas dessas estatísticas criando um código que concatene várias Strings, de maneira que sobrecarregue o pool, gere bastante processamento e consumo de memória.

```
public class Teste {

    public static void main(String[] args) {
        for (int i = 0; i < 1000000; i++) {
            String x = "a" + i;
            System.out.println(x);
        }
    }

}
```

Para analisarmos o resultado do código, vamos rodar o código do `main` através do menu *Run* -> *Profile As* -> *Java Application*.

VERSÕES

Infelizmente o TPTP funciona somente no Windows. Versões para MacOS e Linux são prometidas, mas até hoje estão em desenvolvimento. Uma alternativa paga para esses outros sistemas operacionais é o JProfiler.

CAPÍTULO 23

RESOLUÇÕES DE EXERCÍCIOS

23.1 EXERCÍCIOS 3.3: VARIÁVEIS E TIPOS PRIMITIVOS

1. Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos 15000 reais, em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total no trimestre e a média mensal de gastos. Siga esses passos:

    - Crie uma classe chamada `BalancoTrimestral` com um bloco main, como nos exemplos anteriores;
    - Dentro do `main` (o miolo do programa), declare uma variável inteira chamada `gastosJaneiro` e inicialize-a com 15000;
    - Crie também as variáveis `gastosFevereiro` e `gastosMarco` , inicializando-as com 23000 e 17000, respectivamente, utilize uma linha para cada declaração;
    - Crie uma variável chamada `gastosTrimestre` e inicialize-a com a soma das outras 3 variáveis;
    - Crie uma variável chamada `mediaPorMes` e inicialize-a com a divisão de `gastosTrimestre` por 3.

    - Imprima a variável `gastosTrimestre` .

    Abaixo o código completo:

```
class BalancoTrimestral {
  public static void main(String[] args) {
      int gastosJaneiro = 15000;
      int gastosFevereiro = 23000;
      int gastosMarco = 17000;
      int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco;

      System.out.println("Gasto do trimestre: R$" + gastosTrimestre);
      int mediaPorMes = gastosTrimestre / 3;
      System.out.println("Média mensal: R$" + mediaPorMes);
  }
}
```

23.2 EXERCÍCIOS 3.13: FIXAÇÃO DE SINTAXE

Mais exercícios de fixação de sintaxe. Para quem já conhece um pouco de Java, pode ser muito simples; mas recomendamos fortemente que você faça os exercícios para se acostumar com erros de compilação, mensagens do javac, convenção de código, etc...

Apesar de extremamente simples, precisamos praticar a sintaxe que estamos aprendendo. Para cada exercício, crie um novo arquivo com extensão **.java**, e declare aquele estranho cabeçalho, dando nome a uma classe e com um bloco `main` dentro dele:

```
class ExercicioX {
    public static void main(String[] args) {
        // seu exercício vai aqui
    }
}
```

Não copie e cole de um exercício já existente! Aproveite para praticar.

1. Imprima todos os números de 150 a 300.

   ```
   class ImprimeIntervalo {
       public static void main(String[] args) {
           int i = 150;
           while (i<=300){
               System.out.println(i);
               i++;
           }
       }
   }
   ```

   ou

   ```
   class ImprimeIntervalo {
       public static void main(String[] args) {
           for (int i = 150; i<=300; i++){
               System.out.println(i);
   ```

```
        }
    }
}
```

2. Imprima a soma de 1 até 1000.

```
class ImprimeSoma {
    public static void main(String[] args) {
        int soma = 0;
        int i = 1;
        while (i<=1000){
            soma = soma + i;
            i++;
        }
        System.out.println(i);
    }
}
```

ou

```
class ImprimeSoma {
    public static void main(String[] args) {
        soma = 0;
        for (int i = 1; i<=1000; i++){
            soma = soma + i;
        }
        System.out.println(i);
    }
}
```

3. Imprima todos os múltiplos de 3, entre 1 e 100.

```
class MultiplosDeTresAteCem {
    public static void main (String[] args) {
        for (int i = 1; i < 100; i++ ){
            if (i % 3 == 0)    {
                System.out.println(i);
            }
        }
    }
}
```

ou, entre outras tantas opções, a mais simples:

```
class MultiplosDeTresAteCem {
    public static void main (String[] args) {
        for (int i = 1; i < 100; i += 3 ){
            System.out.println(i);
        }
    }
}
```

4. Imprima os fatoriais de 1 a 10.

O fatorial de um número n é n * (n-1) * (n-2) * ... * 1. Lembre-se de utilizar os parênteses.

O fatorial de 0 é 1

O fatorial de 1 é (0!) * 1 = 1

O fatorial de 2 é (1!) * 2 = 2

O fatorial de 3 é (2!) * 3 = 6

O fatorial de 4 é (3!) * 4 = 24

Faça um for que inicie uma variável n (número) como 1 e fatorial (resultado) como 1 e varia n de 1 até 10:

```
int fatorial = 1;
for (int n = 1; n <= 10; n++) {

}
```

```
class Fatorial {
    public static void main (String[] args) {
        int fatorial = 1;
        for (int n = 1; n <= 10; n++) {
            fatorial = fatorial * n;
            System.out.println("fat(" + n + ") = " + fatorial);
        }
    }
}
```

5. No código do exercício anterior, aumente a quantidade de números que terão os fatoriais impressos, até 20, 30, 40. Em um determinado momento, além desse cálculo demorar, vai começar a mostrar respostas completamente erradas. Por quê?

   Mude de `int` para `long` para ver alguma mudança.

   Resposta:

   Isso acontece porque, a partir de 16!, o valor supera o limite superior do tipo `int`. O tipo `long` consegue armazenar o cálculo dos fatoriais até 21!. Teste!

6. (opcional) Imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc... Para calculá-la, o primeiro elemento vale 0, o segundo vale 1, daí por diante, o n-ésimo elemento vale o (n-1)-ésimo elemento somado ao (n-2)-ésimo elemento (ex: 8 = 5 + 3).

```
class Fibonacci {
    public static void main(String[] args) {
        int anterior = 0;
        int atual = 1;
        while (atual < 100) {
            System.out.println(atual);
            int proximo = anterior + atual;
            anterior = atual;
            atual = proximo;
        }
        System.out.println(atual);
    }
}
```

7. (opcional) Escreva um programa que, dada uma variável `x` com algum valor inteiro, temos um novo `x` de acordo com a seguinte regra:

- se `x` é par, `x = x / 2`
- se `x` é impar, `x = 3 * x + 1`
- imprime `x`

- O programa deve parar quando `x` tiver o valor final de 1. Por exemplo, para `x = 13`, a saída será:

  40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

IMPRIMINDO SEM PULAR LINHA

Um detalhe importante é que uma quebra de linha é impressa toda vez que chamamos `println`. Para não pular uma linha, usamos o código a seguir:

```
System.out.print(variavel);
```

```
class TresNMaisUm {
    public static void main(String[] args) {
        int x = 13;
        System.out.println("Iniciando...\n" + x);
        while (x != 1) {
            if (x % 2 == 0) {
                x = x / 2;
            } else {
                x = 3 * x + 1;
            }
            System.out.println(x);
        }
    }
}
```

8. (opcional) Imprima a seguinte tabela, usando `for` s encadeados:

```
1
2 4
3 6 9
4 8 12 16
n n*2 n*3 .... n*n
```

```
class Triangulo {
    public static void main(String[] args) {
        int numero = 5;
        for (int linha = 1; linha <= numero; linha++) {
            for (int coluna = 1; coluna <= linha; coluna++) {
                System.out.print(linha * coluna + " ");
            }
            System.out.println();
        }
    }
```

```
    }
```

23.3 DESAFIOS 3.14: FIBONACCI

1. Faça o exercício da série de Fibonacci usando apenas duas variáveis.

```
class Desafio {
    public static void main(String[] args) {
        int anterior = 0;
        int atual = 1;
        while (atual < 100) {
            System.out.println(atual);
            atual = anterior + atual;
            anterior = atual - anterior;
        }
        System.out.println(atual);
    }
}
```

23.4 EXERCÍCIOS 4.12: ORIENTAÇÃO A OBJETOS

O modelo da conta a seguir será utilizado para os exercícios dos próximos capítulos.

O objetivo aqui é criar um sistema para gerenciar as contas de um `Banco` . **Os exercícios desse capítulo são extremamente importantes**.

1. Modele uma conta. A ideia aqui é apenas modelar, isto é, só identifique que informações são importantes. Desenhe no papel tudo o que uma `Conta` tem e tudo o que ela faz. Ela deve ter o nome do titular (`String`), o número (`int`), a agência (`String`), o saldo (`double`) e uma data de abertura (`String`). Além disso, ela deve fazer as seguintes ações: saca, para retirar um valor do saldo; deposita, para adicionar um valor ao saldo; calculaRendimento, para devolver o rendimento mensal dessa conta, que é de 10% sobre o saldo.

   Resposta:

   Modelando uma conta

   Toda conta **tem**:

   - String titular;
   - int numero;
   - String agencia;
   - double saldo;

   - String dataDeAbertura;

     Toda conta **faz**:

   - `saca` : retira uma determinada quantia do saldo da conta

- `deposita` : adiciona uma determinada quantia no saldo da conta

- `calculaRendimento` : devolve o quanto essa conta rende por mês

2. Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o `main` . Você deve criar a classe da conta com o nome `Conta` , mas pode nomear como quiser a classe de testes, por exemplo pode chamá-la `TestaConta` , contudo, ela deve necessariamente possuir o método `main` .

   A classe Conta deve conter, além dos atributos mencionados anteriormente, pelo menos os seguintes métodos:

   - `saca` que recebe um `valor` como parâmetro e retira esse valor do saldo da conta
   - `deposita` que recebe um `valor` como parâmetro e adiciona esse valor ao saldo da conta

   - `calculaRendimento` que não recebe parâmetro algum e devolve o valor do saldo multiplicado por 0.1

     Um esboço da classe:

```
class Conta {

    double saldo;
    // seus outros atributos e métodos

    void saca(double valor) {
        // o que fazer aqui dentro?
    }

    void deposita(double valor) {
        // o que fazer aqui dentro?
    }

    double calculaRendimento() {
        // o que fazer aqui dentro?
    }
}
```

     Você pode (e deve) compilar seu arquivo java sem que você ainda tenha terminado sua classe `Conta` . Isso evitará que você receba dezenas de erros de compilação de uma vez só. Crie a classe `Conta` , coloque seus atributos e, antes de colocar qualquer método, compile o arquivo java. O arquivo `Conta.class` será gerado, mas não podemos "executá-lo" já que essa classe não tem um `main` . De qualquer forma, a vantagem é que assim verificamos que nossa classe `Conta` já está tomando forma e está escrita em sintaxe correta.

     Esse é um processo incremental. Procure desenvolver assim seus exercícios, para não descobrir só no fim do caminho que algo estava muito errado.

     Um esboço da classe que possui o `main` :

```
class TestaConta {

    public static void main(String[] args) {
        Conta c1 = new Conta();

        c1.titular = "Hugo";
        c1.numero = 123;
        c1.agencia = "45678-9";
        c1.saldo = 50.0;
        c1.dataDeAbertura = "04/06/2015";

        c1.deposita(100.0);
        System.out.println("saldo atual:" + c1.saldo);
        System.out.println("rendimento mensal:" + c1.calculaRendimento());
    }
}
```

Incremente essa classe. Faça outros testes, imprima outros atributos e invoque os métodos que você criou a mais.

Lembre-se de seguir a convenção java, isso é importantíssimo. Isto é, preste atenção nas maiúsculas e minúsculas, seguindo o seguinte exemplo: `nomeDeAtributo` , `nomeDeMetodo` , `nomeDeVariavel` , `NomeDeClasse` , etc...

Todas as classes no mesmo arquivo?

Você até pode colocar todas as classes no mesmo arquivo e apenas compilar esse arquivo. Ele vai gerar um `.class` para cada classe presente nele.

Porém, por uma questão de organização, é boa prática criar um arquivo `.java` para cada classe. Em capítulos posteriores, veremos também determinados casos nos quais você será **obrigado** a declarar cada classe em um arquivo separado.

Essa separação não é importante nesse momento do aprendizado, mas se quiser ir praticando sem ter que compilar classe por classe, você pode dizer para o `javac` compilar todos os arquivos java de uma vez:

```
javac *.java
```

Abaixo a resposta completa deste item:

```
class Conta {
    String titular;
    int numero;
    String agencia;
    double saldo;
    String dataDeAbertura;

    void saca (double valor) {
```

```
            this.saldo -= valor;
        }

        void deposita (double valor) {
            this.saldo += valor;
        }

        double calculaRendimento() {
            return this.saldo * 0.1;
        }
    }
```

3. Na classe Conta, crie um método `recuperaDadosParaImpressao()` , que não recebe parâmetro mas devolve o texto com todas as informações da nossa conta para efetuarmos a impressão.

   Dessa maneira, você não precisa ficar copiando e colando um monte de `System.out.println()` para cada mudança e teste que fizer com cada um de seus funcionários, você simplesmente vai fazer:

   ```
   Conta c1 = new Conta();
   // brincadeiras com c1....
   System.out.println(c1.recuperaDadosParaImpressao());
   ```

   Veremos mais a frente o método `toString` , que é uma solução muito mais elegante para mostrar a representação de um objeto como `String` , além de não jogar tudo pro `System.out` (só se você desejar).

   O esqueleto do método ficaria assim:

   ```
   class Conta {

       // seus outros atributos e métodos

       String recuperaDadosParaImpressao() {
           String dados = "Titular: " + this.titular;
           dados += "\nNúmero: " + this.numero;
           // imprimir aqui os outros atributos...
           // também pode imprimir this.calculaRendimento()
           return dados;
       }
   }
   ```

   Abaixo a resposta completa deste item:

   ```
   class Conta {
       // outros atributos e métodos...

       String recuperaDadosParaImpressao() {
           String dados = "Titular: " + this.titular;
           dados += "\nNúmero: " + this.numero;
           dados += "\nAgência: " + this.agencia;
           dados += "\nSaldo: R$" + this.saldo;
           dados += "\nData de abertura: " + this.dataDeAbertura;
           return dados;
       }
   }
   ```

4. Na classe de teste dentro do bloco main, construa duas contas com o `new` e compare-os com o `==` .

E se eles tiverem os mesmos atributos? Para isso você vai precisar criar outra referência:

```
Conta c1 = new Conta();
c1.titular = "Danilo";
c1.saldo = 100;

Conta c2 = new Conta();
c2.titular = "Danilo";
c2.saldo = 100;

if (c1 == c2) {
    System.out.println("iguais");
} else {
    System.out.println("diferentes");
}
```

Em ambos os casos, temos `false` como resposta. Isso é porque variáveis guardam apenas as referências! Por mais que dois objetos diferentes tenham as mesmas informações, cada um deles é um objeto à parte.

Você pode ver isso de uma forma simples: se você alterar o `c1`, note que o `c2` não é alterado junto. Cada um é um objeto diferente e cada variável (`c1 e c2`) referencia um deles.

5. Agora, crie duas referências para a **mesma** conta, compare-os com o `==`. Tire suas conclusões. Para criar duas referências pra mesma conta:

```
Conta c1 = new Conta():
c1.titular = "Hugo";
c1.saldo = 100;

c2 = c1;
```

O que acontece com o `if` do exercício anterior?

Resposta:

Agora sim, obtemos true. Isso porque, de fato, ambas as variáveis têm referências para o mesmo objeto. Verifique: mude o titular da `c1` para *Mariana* e imprima `c2.titular`. Você vai notar que o nome mudou!

6. (opcional) Em vez de utilizar uma `String` para representar a data, crie uma outra classe, chamada `Data`. Ela possui 3 campos `int`, para dia, mês e ano. Faça com que sua conta passe a usá-la. (é parecido com o último exemplo da explicação, em que a `Conta` passou a ter referência para um `Cliente`).

```
class Conta {
    Data dataDeAbertura; // qual é o valor default aqui?
    // seus outros atributos e métodos
}

class Data {
    int dia;
    int mes;
```

```java
    int ano;
}
```

Modifique sua classe `TestaConta` para que você crie uma `Data` e atribua ela a `Conta` :

```java
Conta c1 = new Conta();
//...
Data data = new Data(); // ligação!
c1.dataDeAbertura = data;
```

Faça o desenho do estado da memória quando criarmos um `Conta` . Uma possibilidade é o arquivo `Data.java` ficar assim:

```java
class Data {
    int dia;
    int mes;
    int ano;

    void preencheData (int dia, int mes, int ano) {
        this.dia = dia;
        this.mes = mes;
        this.ano = ano;
    }
}
```

No arquivo `Conta.java` , altere o atributo para a data:

```java
class Conta {
    // outros atributos
    Data dataDeAbertura;

    // metodos
}
```

Finalmente no arquivo `TestaConta.java` ...

```java
class TestaConta {
    public static void main(String[] args) {
        Conta c1 = new Conta();
        c1.titular = "Hugo";
        c1.saldo = 50;
        c1.deposita(100);

        // adicionando a data como tipo
        c1.dataDeAbertura = new Data();
        c1.dataDeAbertura.preencheData(1, 7, 2009);

        System.out.println(c1.recuperaDadosParaImpressao());
    }
}
```

7. (opcional) Modifique seu método `recuperaDadosParaImpressao` para que ele devolva o valor da `dataDeAbertura` daquela `Conta` :

```java
class Conta {

    // seus outros atributos e métodos
    Data dataDeAbertura;
```

```
    String recuperaDadosParaImpressao() {
        String dados = "\nTitular: " + this.titular;
        // imprimir aqui os outros atributos...

        dados += "\nDia: " + this.dataDeAbertura.dia;
        dados += "\nMês: " + this.dataDeAbertura.mes;
        dados += "\nAno: " + this.dataDeAbertura.ano;
        return dados;
    }
}
```

Teste-o. O que acontece se chamarmos o método `recuperaDadosParaImpressao` antes de atribuirmos uma data para esta `Conta` ?

Resposta: dia, mês e ano serão apresentados com o valor 0.

```
class TestaConta {
public static void main(String[] args) {
    Conta c1 = new Conta();
    c1.titular = "Hugo";
    c1.agencia = "12-x";
    c1.numero = 557890;
    c1.saldo = 50;
    c1.deposita(100);
    // adicionando a data como tipo
    c1.dataDeAbertura = new Data();
    //chamando o método `recuperaDadosParaImpressao` antes de atribuirmos uma data para esta `Con
ta`
    System.out.println(c1.recuperaDadosParaImpressao());
    c1.dataDeAbertura.preencheData(1, 7, 2009);
}
}
```

8. (opcional) O que acontece se você tentar acessar um atributo diretamente na classe? Como, por exemplo:

```
    Conta.saldo = 1234;
```

Esse código faz sentido? E este:

```
    Conta.calculaRendimento();
```

Faz sentido perguntar para o esquema da Conta seu valor anual?

9. (opcional-avançado) Crie um método na classe `Data` que devolva o valor formatado da data, isto é, devolva uma String com "dia/mes/ano". Isso para que o método `recuperaDadosParaImpressao` da classe `Conta` possa ficar assim:

```
class Conta {
    // atributos e metodos

    String recuperaDadosParaImpressao() {
        // imprime outros atributos...
        dados += "\nData de abertura: " + this.dataDeAbertura.formatada();
        return dados;
    }
}
```

Resposta:

```
class Data {
    // atributos e método preencheData

    String formatada() {
        return this.dia + "/" + this.mes + "/" + this.ano;
    }
}
```


23.5 DESAFIOS 4.13

1. Um método pode chamar ele mesmo. Chamamos isso de **recursão**. Você pode resolver a série de Fibonacci usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da seguinte maneira:

```
Fibonacci fibonacci = new Fibonacci();
for (int i = 1; i <= 6; i++) {
    int resultado = fibonacci.calculaFibonacci(i);
    System.out.println(resultado);
}
```

Aqui imprimirá a sequência de Fibonacci até a sexta posição, isto é: 1, 1, 2, 3, 5, 8.

Este método `calculaFibonacci` não pode ter nenhum laço, só pode chamar ele mesmo como método. Pense nele como uma função, que usa a própria função para calcular o resultado.

Resposta:

```
class Fibonacci{

    public int calculaFibonacci(int n) {
        if (n <= 2) {
            return 1;
        } else {
            return calculaFibonacci(n-1) + calculaFibonacci(n-2);
        }
    }
```

```
}
```

2. Por que o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se usa um laço)?

   Resposta:

   Dessa forma, o código fica muito mais lento porque ele não consegue aproveitar os fibonaccis já calculados anteriormente. E, pior ainda, ele abre o cálculo de fibonaccis exponencialmente porque, para calcular o fibonacci de um número, é preciso somar os dois anteriores.

3. Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o **operador condicional ternário**. (ternary operator)

   Resposta:

```
public static int calculaFibonacci(int n) {
    return (n <= 2) ? 1 : calculaFibonacci(n-1) + calculaFibonacci(n-2);
}
```

23.6 EXERCÍCIOS 5.8: ENCAPSULAMENTO, CONSTRUTORES E STATIC

1. Adicione o modificador de visibilidade (`private` , se necessário) para cada atributo e método da classe `Conta` . Tente criar uma `Conta` no `main` e modificar ou ler um de seus atributos privados. O que acontece?

```
public class Conta {
    private String titular;
    private int numero;
    private String agencia;
    private double saldo;
    private Data dataDeAbertura;

    public void saca(double valor) {...}

    public void deposita(double valor) {...}

    public double calculaRendimento() {...}

    public String recuperaDadosParaImpressao() {...}
}
```

2. Crie apenas os getters e setters necessários da sua classe `Conta` . Pense sempre se é preciso criar cada um deles. Por exemplo:

```
class Conta {
    private String titular;

    // ...

    public String getTitular() {
        return this.titular;
```

```java
    }

    public void setTitular(String titular) {
        this.titular = titular;
    }
}
```

Não copie e cole! Aproveite para praticar sintaxe. Logo passaremos a usar o Eclipse e aí sim teremos procedimentos mais simples para este tipo de tarefa.

Repare que o método `calculaRendimento` parece também um getter. Aliás, seria comum alguém nomeá-lo de `getRendimento`. Getters não precisam apenas retornar atributos. Eles podem trabalhar com esses dados.

Resposta completa para este item:

```java
public class Conta {
    private String titular;
    private int numero;
    private String agencia;
    private double saldo;
    private String dataDeAbertura;

    public double calculaRendimento() {
        return this.saldo * 0.1;
    }

    public String getTitular() {
        return this.titular;
    }

    public void setTitular (String titular) {
        this.titular = titular;
    }

    public int getNumero() {
        return this.numero;
    }

    public void setNumero (int numero) {
        this.numero = numero;
    }

    public String getAgencia() {
        return this.agencia;
    }

    public void setAgencia (String agencia) {
        this.agencia = agencia;
    }

    public double getSaldo() {
        return this.saldo;
    }

    public void deposita (double valor) {
        this.saldo += valor;
    }
```

```
        public void saca (double valor) {
            this.saldo -= valor;
        }

        public String getDataDeAbertura() {
            return this.dataDeAbertura;
        }

        public void setDataDeAbertura (String dataDeAbertura) {
            this.dataDeAbertura = dataDeAbertura;
        }
    }
```

3. Altere suas classes que acessam e modificam atributos de uma `Conta` para utilizar os getters e setters recém criados.

   Por exemplo, onde você encontra:

```
    c.titular = "Batman";
    System.out.println(c.titular);
```

   passa para:

```
    c.setTitular("Batman");
    System.out.println(c.getTitular());
```

   Resposta completa para este item:

```
    class TestaConta {
        public static void main(String[] args) {
            Conta c1 = new Conta("Hugo");
            c1.setNumero(123);
            c1.deposita(50);
            c1.setDataDeAbertura(new Data(1, 7, 2009));

            System.out.println(c1.recuperaDadosParaImpressao());
        }
    }
```

4. Faça com que sua classe `Conta` possa receber, opcionalmente, o nome do titular da `Conta` durante a criação do objeto. Utilize construtores para obter esse resultado.

   Dica: utilize um construtor sem argumentos também, para o caso de a pessoa não querer passar o titular da `Conta` .

   Seria algo como:

```
    class Conta {
        public Conta() {
            // construtor sem argumentos
        }

        public Conta(String titular) {
            // construtor que recebe o titular
        }
    }
```

Por que você precisa do construtor sem argumentos para que a passagem do nome seja opcional?

Resposta: A partir do momento que declaramos um construtor, o construtor default não é mais fornecido, por isso se quisermos ter a passagem do nome como opcional, teremos que ter duas versões de construtores: uma que não exige nada como parâmetro e outra que exige uma String.

```
public class Conta {
    // atributos

    public Conta() {}

    public Conta(String titular) {
        this.titular = titular;
    }

    // métodos
}
```

5. (opcional) Adicione um atributo na classe `Conta` de tipo `int` que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo `Conta` . A primeira `Conta` instanciada tem identificador 1, a segunda 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.

   Crie um getter para o identificador. Devemos ter um setter?

   Resposta:

```
public class Conta {
    private int identificador;
    private static int proximoIdentificador;

    public Conta(String titular) {
        this.titular = titular;
        this.identificador = proximoIdentificador++;
    }

    public int getIdentificador() {
        return this.identificador;
    }

    // restante da classe
}
```

   Não faz sentido que o identificador tenha um setter já que, pela lógica da aplicação, o `identificador` é um número único para cada funcionário no sistema.

6. (opcional) Como garantir que datas como 31/2/2021 não sejam aceitas pela sua classe `Data` ?

   Resposta: Você pode definir um construtor na classe Data que exija dia, mes e ano. Este construtor invoca o método `preencheData` o qual invoca o método `isDataViavel` que faz a validação das datas válidas. Nesse momento, a única forma de indicar que houve um erro que você aprendeu é imprimir uma mensagem no terminal avisando, mas, mais para a frente, veremos uma forma muito mais elegante de tratar esses casos.

```java
public class Data {

    private int dia;
    private int mes;
    private int ano;

    public Data(int dia, int mes, int ano) {
        this.preencheData(dia, mes, ano);
    }

    private boolean isDataViavel(int dia, int mes, int ano) {
        if (dia <= 0 || mes <= 0) {
            return false;
        }

        int ultimoDiaDoMes = 31; // por padrao são 31 dias

        if (mes == 4 || mes == 6 || mes == 9 || mes == 11) {
            ultimoDiaDoMes = 30;
        } else if (mes == 2) {
            if (ano % 4 == 0) {
                ultimoDiaDoMes = 29;
            } else {
                ultimoDiaDoMes = 28;
            }
        }
        if (dia > ultimoDiaDoMes) {
            return false;
        }

        return true;
    }

    void preencheData(int dia, int mes, int ano) {
        if (!isDataViavel(dia, mes, ano)) {
            System.out.println("A data " + dia + "/" + mes + "/" + ano + " não existe!");
        } else {
            this.dia = dia;
            this.mes = mes;
            this.ano = ano;
        }
    }

    String formatada() {
        return this.dia + "/" + this.mes + "/" + this.ano;
    }
}
```

Faça testes com datas válidas e inválidas:

```java
class TestaDataAberturaDaConta {

public static void main(String[] args) {
    Conta c1 = new Conta();
    c1.setTitular("Hugo"); ;
    c1.setAgencia("12-x");;
    c1.setNumero(557890); ;
    c1.deposita(50);;
    c1.deposita(100);
    // adicionando a data como tipo
    c1.setDataDeAbertura(new Data(31, 2, 2021)); // teste também com datas válidas
    System.out.println(c1.recuperaDadosParaImpressao());
```

```
    }
}
```

1. (opcional) Suponha que temos a classe `PessoaFisica` que tem um CPF como atributo. Como garantir que pessoa física alguma tenha CPF invalido, nem seja criada `PessoaFisica` sem cpf inicial?

   Resposta: (Suponha que já existe um algoritmo de validação de cpf: basta passar o cpf por um método `valida(String x)...`)

   Você pode fazer a validação ser chamada no construtor e, por ora, imprimir a mensagem no console. No capítulo 11 veremos uma forma de realmente impedir a criação do objeto, caso essa validação não passe.

23.7 DESAFIOS 5.9

1. Porque esse código não compila?

   ```
   class Teste {
       int x = 37;
       public static void main(String [] args) {
           System.out.println(x);
       }
   }
   ```

   Resposta: O `main` é um método estático, isto é: ele não é do objeto, é da classe. Já o atributo `x` não tem a palavra `static` e, portanto, é do objeto.

   Para rodar o `main` , não há necessidade (ou garantia) de que teremos um objeto do tipo `Teste` , então não há garantia de que o `x` sequer existirá.

2. Imagine que tenha uma classe `FabricaDeCarro` e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em Java, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso? (pesquise: singleton design pattern)

Seus livros de tecnologia parecem do século passado?

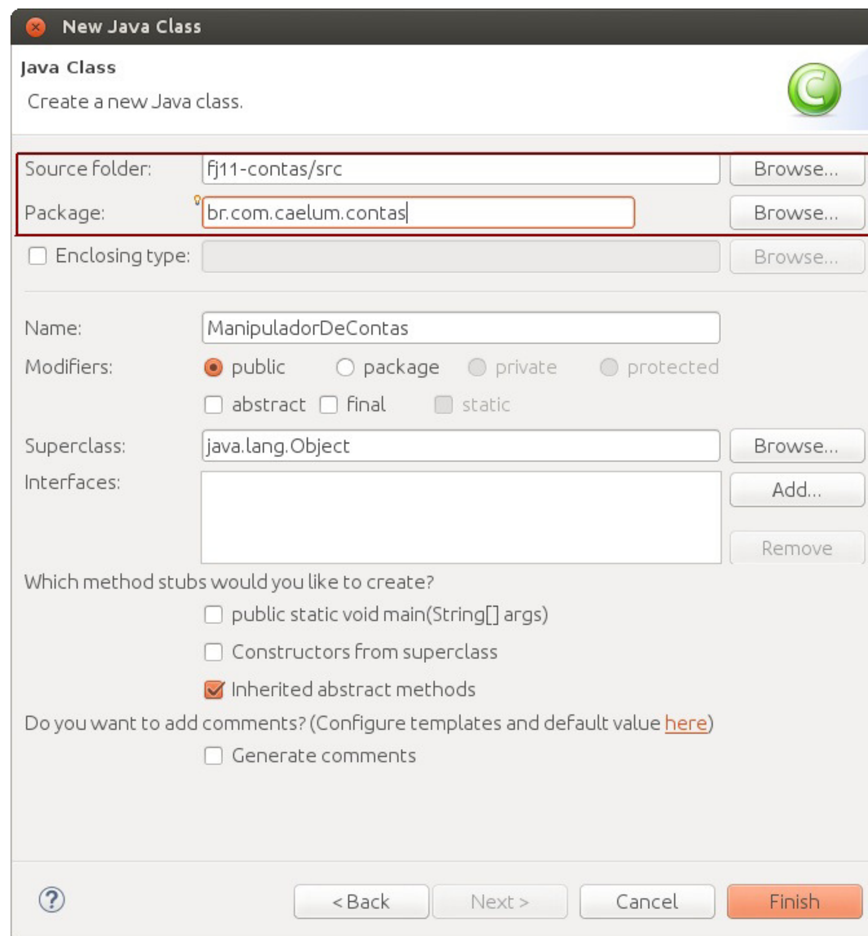


Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.]

23.8 EXERCÍCIOS 8.9: MOSTRANDO OS DADOS DA CONTA NA TELA

1. Crie a classe `ManipuladorDeContas` dentro do pacote `br.com.caelum.contas` . Repare que os pacotes `br.com.caelum.contas.main` e `br.com.caelum.contas.modelo` são subpacotes do pacote `br.com.caelum.contas` , portanto o pacote `br.com.caelum.contas` já existe. Para criar a classe neste pacote, basta selecioná-lo na janela de criação da classe:



A classe `ManipuladorDeContas` fará a ligação da `Conta` com a tela, por isso precisaremos declarar um atributo do tipo `Conta` .

2. Crie o método `criaConta` que recebe como parâmetro um objeto do tipo `Evento` . Instancie uma conta para o atributo `conta` e coloque os valores de `numero` , `agencia` e `titular` . Algo como:

```
public class ManipuladorDeContas {

private Conta conta;

public void criaConta(Evento evento){
    this.conta = new Conta();
    this.conta.setTitular("Batman");
    // faça o mesmo para os outros atributos
}
```

3. Com a conta instanciada, agora podemos implementar as funcionalidades de saque e depósito. Crie o método `deposita` que recebe um `Evento` , que é a classe que retorna os dados da tela nos tipos que precisamos. Por exemplo, se quisermos o valor a depositar sabemos que ele é do tipo `double` e que o nome do campo na tela é `valor` então podemos fazer:

```
public void deposita(Evento evento){
```

```
        double valorDigitado = evento.getDouble("valor");
        this.conta.deposita(valorDigitado);
    }
```

4. Crie agora o método `saca`. Ele também deve receber um `Evento` nos mesmos moldes do `deposita`.

```
    public void saca(Evento evento){
        double valorDigitado = evento.getDouble("valor");
        this.conta.saca(valorDigitado);
    }
```

5. Precisamos agora testar nossa aplicação, crie a classe `TestaContas` dentro do pacote `br.com.caelum.contas` com um `main`. Nela vamos chamar o `main` da classe `TelaDeContas` que mostrará a tela de nosso sistema. Não se esqueça de fazer o import desta classe!

```
import br.com.caelum.javafx.api.main.TelaDeContas;

public class TestaContas {

    public static void main(String[] args) {
        TelaDeContas.main(args);
    }
}
```

Rode a aplicação, crie a conta e tente fazer as operações de saque e depósito. Tudo deve funcionar normalmente.

23.9 EXERCÍCIOS 9.7: HERANÇA E POLIMORFISMO

1. Vamos ter mais de um tipo de conta no nosso sistema então vamos precisar de uma nova tela para cadastrar os diferentes tipos de conta. Essa tela já está pronta e para utilizá-la só precisamos alterar a classe que estamos chamando no método `main()` no `TestaContas.java`:

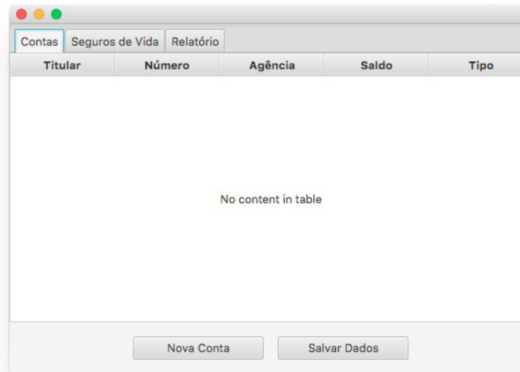
```
package br.com.caelum.contas.main;

import br.com.caelum.javafx.api.main.SistemaBancario;

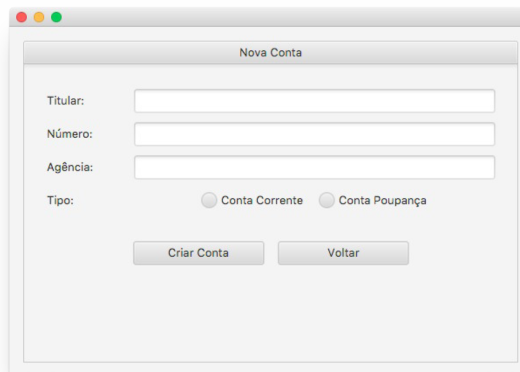
public class TestaContas {

    public static void main(String[] args) {
        SistemaBancario.mostraTela(false);
        // TelaDeContas.main(args);
    }
}
```

2. Ao rodar a classe `TestaContas` agora, teremos a tela abaixo:



Vamos entrar na tela de criação de contas para vermos o que precisamos implementar para que o sistema funcione. Para isso, clique no botão **Nova Conta**. A seguinte tela aparecerá:



Podemos perceber que além das informações que já tínhamos na conta, temos agora o tipo: se queremos uma conta corrente ou uma conta poupança. Vamos então criar as classes correspondentes.

- Crie a classe `ContaCorrente` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`
- Crie a classe `ContaPoupanca` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`

3. Precisamos pegar os dados da tela para conseguirmos criar a conta correspondente. No `ManipuladorDeContas` vamos alterar o método `criaConta`. Atualmente, apenas criamos uma nova conta com os dados direto no código. Vamos fazer com que agora os dados sejam recuperados da tela para colocarmos na nova conta, faremos isso utilizando o objeto `evento`:

```
public void criaConta(Evento evento) {
      this.conta = new Conta();
      this.conta.setAgencia(evento.getString("agencia"));
      this.conta.setNumero(evento.getInt("numero"));
      this.conta.setTitular(evento.getString("titular"));
  }
```

Mas precisamos dizer qual tipo de conta que queremos criar! Devemos então recuperar o tipo da conta escolhido e criar a conta correspondente. Para isso, ao invés de criar um objeto do tipo 'Conta', vamos usar o método `getSelecionadoNoRadio` do objeto `evento` para pegar o tipo, fazer um `if` para verificar esse tipo e só depois criar o objeto do tipo correspondente. Após essas mudanças, o método `criaConta` ficará como abaixo:

```
public void criaConta(Evento evento) {
    String tipo = evento.getSelecionadoNoRadio("tipo");
    if (tipo.equals("Conta Corrente")) {
    this.conta = new ContaCorrente();
    } else if (tipo.equals("Conta Poupança")) {
    this.conta = new ContaPoupanca();
   }
    this.conta.setAgencia(evento.getString("agencia"));
    this.conta.setNumero(evento.getInt("numero"));
    this.conta.setTitular(evento.getString("titular"));
}
```

4. Apesar de já conseguirmos criar os dois tipos de contas, nossa lista não consegue exibir o tipo de cada conta na lista da tela inicial. Para resolver isso, podemos criar um método `getTipo` em cada uma de nossas contas fazendo com que a conta corrente devolva a string "Conta Corrente" e a conta poupança devolva a string "Conta Poupança":

```
public class ContaCorrente extends Conta {
    public String getTipo() {
        return "Conta Corrente";
    }
}

public class ContaPoupanca extends Conta {
    public String getTipo() {
        return "Conta Poupança";
    }
}
```

5. Altere os métodos `saca` e `deposita` para buscarem o campo `valorOperacao` ao invés de apenas `valor` na classe `ManipuladorDeContas`.

6. Vamos mudar o comportamento da operação de saque de acordo com o tipo de conta que estiver sendo utilizada. Na classe `ManipuladorDeContas` vamos alterar o método `saca` para tirar 10 centavos de cada saque em uma conta corrente:

```
public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    if (this.conta.getTipo().equals("Conta Corrente")){
        this.conta.saca(valor + 0.10);
    } else {
```

```
        this.conta.saca(valor);
    }
}
```

Ao tentarmos chamar o método `getTipo` , o Eclipse reclamou que esse método não existe na classe `Conta` apesar de existir nas classes filhas. Como estamos tratando todas as contas genericamente, só conseguimos acessar os métodos da classe mãe. Vamos então colocá-lo na classe `Conta` :

```
public class Conta {
    public String getTipo() {
        return "Conta";
    }
}
```

7. Agora o código compila mas temos um outro problema. A lógica do nosso saque vazou para a classe `ManipuladorDeContas` . Se algum dia precisarmos alterar o valor da taxa no saque, teríamos que mudar em todos os lugares onde fazemos uso do método `saca` . Esta lógica deveria estar encapsulada dentro do método `saca` de cada conta. Vamos então sobrescrever o método dentro da classe `ContaCorrente` :

```
public class ContaCorrente extends Conta {
    @Override
    public void saca(double valor) {
        this.saldo -= (valor + 0.10);
    }

  // restante da classe
}
```

Repare que, para acessar o atributo saldo herdado da classe `Conta` , **você vai precisar mudar o modificador de visibilidade de saldo para `protected`** .

Agora que a lógica está encapsulada, podemos corrigir o método `saca` da classe `ManipuladorDeContas` :

```
public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    this.conta.saca(valor);
}
```

Perceba que agora tratamos a conta de forma genérica!

8. Rode a classe `TestaContas` , adicione uma conta de cada tipo e veja se o tipo é apresentado corretamente na lista de contas da tela inicial.

Agora, clique na conta corrente apresentada na lista para abrir a tela de detalhes de contas. Teste as operações de saque e depósito e perceba que a conta apresenta o comportamento de uma conta corrente conforme o esperado.

E se tentarmos realizar uma transferência da conta corrente para a conta poupança? O que acontece?

9. Vamos começar implementando o método `transfere` na classe `Conta` :

```
public void transfere(double valor, Conta conta) {
    this.saca(valor);
    conta.deposita(valor);
}
```

Também precisamos implementar o método `transfere` na classe `ManipuladorDeContas` para fazer o vínculo entre a tela e a classe `Conta` :

```
public void transfere(Evento evento) {
    Conta destino = (Conta) evento.getSelecionadoNoCombo("destino");
    conta.transfere(evento.getDouble("valorTransferencia"), destino);
}
```

Rode de novo a aplicação e teste a operação de transferência.

10. Considere o código abaixo:

```
Conta c = new Conta();
ContaCorrente cc = new ContaCorrente();
ContaPoupanca cp = new ContaPoupanca();
```

Se mudarmos esse código para:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual é a utilidade disso? Realmente, essa não é a maneira mais útil do polimorfismo. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é, como fazemos na classe `ManipuladorDeContas` .

É **extremamente importante** perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto, não importando como nos referimos a ele.

11. (Opcional) A nossa classe `Conta` devolve a palavra "Conta" no método `getTipo` . Use a palavra chave `super` nos métodos `getTipo` reescritos nas classes filhas, para não ter de reescrever a palavra "Conta" ao devolver os tipos "Conta Corrente" e "Conta Poupança".

```
class ContaCorrente extends Conta {

    public String getTipo() {
        return super.getTipo() + " Corrente";
    }
}
```

E, também

```
class ContaPoupanca extends Conta {

    public String getTipo() {
        return super.getTipo() + " Poupança";
```

```
    }
    //...
}
```

12. (Opcional) Se você precisasse criar uma classe `ContaInvestimento`, e seu método `saca` fosse complicadíssimo, você precisaria alterar a classe `ManipuladorDeContas`?

    Resposta: Não! Essa é a vantagem do polimorfismo: qualquer coisa que **seja uma** Conta pode ser passada para o método `saca`. A complexidade fica isolada, dentro de cada classe.

23.10 EXERCÍCIOS 11.5: INTERFACES

1. Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso vamos criar uma interface no pacote `br.com.caelum.contas.modelo` do nosso projeto `fj11-contas` já existente:

```
public interface Tributavel {
  public double getValorImposto();
}
```

   Lemos essa interface da seguinte maneira: "todos que quiserem ser *tributável* precisam saber retornar *o valor do imposto*, devolvendo um double".

   Alguns bens são tributáveis e outros não, `ContaPoupanca` não é tributável, já para `ContaCorrente` você precisa pagar 1% da conta e o `SeguroDeVida` tem uma taxa fixa de 42 reais mais 2% do valor do seguro.

   Aproveite o Eclipse! Quando você escrever `implements Tributavel` na classe `ContaCorrente`, o *quick fix* do Eclipse vai sugerir que você reescreva o método; escolha essa opção e, depois, preencha o corpo do método adequadamente:

```
public class ContaCorrente extends Conta implements Tributavel {

    // outros atributos e métodos

    public double getValorImposto() {
        return this.getSaldo() * 0.01;
    }
}
```

   Crie a classe `SeguroDeVida`, aproveitando novamente do Eclipse, para obter:

```
public class SeguroDeVida implements Tributavel {
    private double valor;
    private String titular;
    private int numeroApolice;

    public double getValorImposto() {
        return 42 + this.valor * 0.02;
    }

// getters e setters para os atributos
```

```
}
```

Além disso, escreva o método `getTipo` para que o tipo do produto apareça na interface gráfica:

```
public String getTipo(){
    return "Seguro de Vida";
}
```

2. Vamos criar a classe `ManipuladorDeSeguroDeVida` dentro do pacote `br.com.caelum.contas` para vincular a classe `SeguroDeVida` com a tela de criação de seguros. Esta classe deve ter um atributo do tipo `SeguroDeVida` .

   Crie também o método `criaSeguro` que deve receber um parâmetro do tipo `Evento` para conseguir obter os dados da tela. Você deve pegar os parâmetros "numeroApolice" do tipo `int` , "titular" do tipo `String` e "valor" do tipo `double` .

   O código final deve ficar parecido com o código abaixo:

```
package br.com.caelum.contas;

import br.com.caelum.contas.modelo.SeguroDeVida;
import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeSeguroDeVida {

  private SeguroDeVida seguroDeVida;

  public void criaSeguro(Evento evento){
        this.seguroDeVida = new SeguroDeVida();
        this.seguroDeVida.setNumeroApolice(evento.getInt("numeroApolice"));
        this.seguroDeVida.setTitular(evento.getString("titular"));
        this.seguroDeVida.setValor(evento.getDouble("valor"));
  }
}
```

3. Execute a classe `TestaContas` e tente cadastrar um novo seguro de vida. O seguro cadastrado deve aparecer na tabela de seguros de vida.

4. Queremos agora saber qual o valor total dos impostos de todos os tributáveis. Vamos então criar a classe `ManipuladorDeTributaveis` dentro do pacote `br.com.caelum.contas` . Crie também o método `calculaImpostos` que recebe um parâmetro do tipo `Evento` :

```
package br.com.caelum.contas;

import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeTributaveis {

  public void calculaImpostos(Evento evento){
    // aqui calcularemos o total
  }
}
```

5. Agora que criamos o tributavel, vamos habilitar a última aba de nosso sistema. Altere a classe

`TestaContas` para passar o valor `true` na chamada do método `mostraTela`. Observe que agora que temos o seguro de vida funcionando, a tela de relatório já consegue imprimir o valor dos impostos individuais de cada tipo de *Tributavel*.

6. No método `calculaImpostos` precisamos buscar os valores de impostos de cada `Tributavel` e somá-los. Para saber a quantidade de tributáveis, a classe `Evento` possui um método chamado `getTamanhoDaLista` que deve receber o nome da lista desejada, no caso "listaTributaveis". Existe também um outro método que retorna um `Tributavel` de uma determinada posição de uma lista, onde precisamos passar o nome da lista e o índice do elemento. Precisamos percorrer a lista inteira, passando por cada posição então utilizaremos um `for` para isto.

```
package br.com.caelum.contas;

import br.com.caelum.contas.modelo.Tributavel;
import br.com.caelum.javafx.api.util.Evento;

public class ManipuladorDeTributaveis {

      private double total;

  public void calculaImpostos(Evento evento){
        total = 0;
        int tamanho = evento.getTamanhoDaLista("listaTributaveis");
        for (int i = 0; i < tamanho; i++) {
          Tributavel t = evento.getTributavel("listaTributaveis", i);
          total += t.getValorImposto();
        }
  }

  public double getTotal() {
        return total;
  }
}
```

Repare que, de dentro do `ManipuladorDeTributaveis`, você não pode acessar o método `getSaldo`, por exemplo, pois você não tem a garantia de que o `Tributavel` que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface `Tributavel`.

É interessante enxergar que as interfaces (como aqui, no caso, `Tributavel`) costumam ligar classes muito distintas, unindo-as por uma característica que elas tem em comum. No nosso exemplo, `SeguroDeVida` e `ContaCorrente` são entidades completamente distintas, porém ambas possuem a característica de serem tributáveis.

Se amanhã o governo começar a tributar até mesmo `PlanoDeCapitalizacao`, basta que essa classe implemente a interface `Tributavel`! Repare no grau de desacoplamento que temos: a classe `GerenciadorDeImpostoDeRenda` nem imagina que vai trabalhar como `PlanoDeCapitalizacao`. Para ela, o único fato que importa é que o objeto respeite o contrato de um tributável, isso é, a

interface `Tributavel` . Novamente: programe voltado à interface, não à implementação.

Quais os benefícios de manter o código com baixo acoplamento?

Resposta:

Quanto menos acoplado o código, mais fácil é sua manutenção, já que alterar uma classe não deve atrapalhar o funcionamento das outras. Note que o uso de interfaces cria uma ligação entre tipos que permite o **polimorfismo**, mas é bem menos intrusivo do que a herança: não é possível reaproveitar código da mãe.

Por um lado, isso pode parecer negativo e, por vezes, teremos um trecho de código repetido. Mas a certeza de que, ao mudar uma classe, não afetaremos as outras, é muito confortável. Para usar interfaces **e** evitar a repetição, procure pelo conceito de **composição**.

7. (opcional) Crie a classe `TestaTributavel` com um método `main` para testar o nosso exemplo:

```
public class TestaTributavel {

    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        System.out.println(cc.getValorImposto());

        // testando polimorfismo:
        Tributavel t = cc;
        System.out.println(t.getValorImposto());
    }
}
```

Tente chamar o método `getSaldo` através da referência `t` , o que ocorre? Por quê?

A linha em que atribuímos `cc` a um `Tributavel` é apenas para você enxergar que é possível fazê-lo. Nesse nosso caso, isso não tem uma utilidade. Essa possibilidade foi útil no exercício anterior.

Resposta: Apesar de ser um objeto do tipo `ContaCorrente` , ao chamarmos ele de `Tributavel` , apenas garantimos para o compilador que aquele objeto tem os métodos que **todo** `Tributavel` tem. E como o compilador do Java só trabalha com certezas, ele só permite chamar os métodos definidos no tipo da variável.

23.11 EXERCÍCIOS OPCIONAIS 11.6

Atenção: caso você faça esse exercício, faça isso num projeto à parte `conta-interface` já que usaremos a `Conta` como classe em exercícios futuros.

1. (Opcional) Transforme a classe `Conta` em uma interface.

```
public interface Conta {
    public double getSaldo();
    public void deposita(double valor);
    public void saca(double valor);
    public void atualiza(double taxaSelic);
}
```

Adapte `ContaCorrente` e `ContaPoupanca` para essa modificação:

```
public class ContaCorrente implements Conta {
    // ...
}
```

```
public class ContaPoupanca implements Conta {
    // ...
    }
```

Algum código vai ter de ser copiado e colado? Isso é tão ruim?

Ao fim desse exercício você terá os seguintes códigos:

```
public interface Conta {

    public abstract void deposita(double valor);
    public abstract void saca(double valor);
    public abstract double getSaldo();
    public abstract void atualiza(double taxa);
}
```

E as classes que implementam `Conta` :

```
public class ContaCorrente implements Conta, Tributavel {
```

```
    private double saldo;

    @Override
    public void deposita(double valor) {
        this.saldo += valor;
    }

    @Override
    public void saca(double valor) {
        this.saldo -= valor;
    }

    @Override
    public double getSaldo() {
        return this.saldo;
    }

    @Override
    public void atualiza(double taxa) {
        this.saldo += this.saldo * taxa * 2;
    }

    @Override
    public double calculaTributos() {
        return this.saldo * 0.01;
    }
}
```

```
public class ContaPoupanca implements Conta {
    private double saldo;

    @Override
    public void atualiza(double taxa) {
        this.saldo += this.saldo * taxa * 3;
    }

    @Override
    public void deposita(double valor) {
        this.saldo += (valor - 0.1);
    }

    @Override
    public void saca(double valor) {
        this.saldo -= valor;
    }

    @Override
    public double getSaldo() {
        return this.saldo;
    }
}
```

2. (Opcional) Às vezes, é interessante criarmos uma interface que herda de outras interfaces: essas, são chamadas subinterfaces. Essas, nada mais são do que um agrupamento de obrigações para a classe que a implementar.

```
public interface ContaTributavel extends Conta, Tributavel {
}
```

Dessa maneira, quem for implementar essa nova interface precisa implementar todos os métodos

herdados das suas superinterfaces (e talvez ainda novos métodos declarados dentro dela):

```
public class ContaCorrente implements ContaTributavel {
  // métodos
}

Conta c = new ContaCorrente();
Tributavel t = new ContaCorrente();
```

Repare que o código pode parecer estranho, pois a interface não declara método algum, só herda os métodos abstratos declarados nas outras interfaces.

Ao mesmo tempo que uma interface pode herdar de mais de uma outra interface, classes só podem possuir uma classe mãe (herança simples).

Resposta:

Podemos criar a interface ContaTributavel, que é uma `Conta` e também é `Tributavel`. Como as definições dos métodos já estão nas duas interfaces originais, a declaração da nova fica, simplesmente:

```
public interface ContaTributavel extends Conta, Tributavel {
}
```

E, então, alteramos também a ContaCorrente, que passa a implementar apenas essa nova interface:

```
public class ContaCorrente implements ContaTributavel {
    // restante da classe
    // exatamente igual à do exercício anterior
}
```

23.12 EXERCÍCIOS 12.11: EXCEÇÕES

1. Na classe `Conta`, modifique o método `deposita(double x)`: Ele deve lançar uma exception chamada `IllegalArgumentException`, que já faz parte da biblioteca do Java, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

```
public void deposita(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException();
    } else {
        this.saldo += valor;
    }
}
```

2. Rode a aplicação, cadastre uma conta e tente depositar um valor negativo. O que acontece?

   Resposta: Uma `IllegalArgumentException` é lançada quando tentamos depositar um valor inválido, isto é, o próprio método `deposita` se defende de alguém que queira fazer algo errado.

3. Ao lançar a `IllegalArgumentException`, passe via construtor uma mensagem a ser exibida.

Lembre que a `String` recebida como parâmetro é acessível depois via o método `getMessage()` herdado por todas as `Exceptions` .

```
public void deposita(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException("Você tentou depositar" +
                                           " um valor negativo");
    } else {
        this.saldo += valor;
    }
}
```

Rode a aplicação novamente e veja que agora a mensagem aparece na tela.

4. Faça o mesmo para o método `saca` da classe `ContaCorrente` , afinal o cliente também não pode sacar um valor negativo!

5. Vamos validar também que o cliente não pode sacar um valor maior do que o saldo disponível em conta. Crie sua própria `Exception` , `SaldoInsuficienteException` . Para isso, você precisa criar uma classe com esse nome que seja filha de `RuntimeException` .

```
public class SaldoInsuficienteException extends RuntimeException {

}
```

No método `saca` da classe `ContaCorrente` vamos utilizar esta nova `Exception` :

```
@Override
public void saca(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException("Você tentou sacar um valor negativo");
    }
    if (this.saldo < valor) {
        throw new SaldoInsuficienteException();
    }
    this.saldo -= (valor + 0.10);
}
```

Atenção: nem sempre é interessante criarmos um novo tipo de exception! Depende do caso. Neste aqui, seria melhor ainda utilizarmos `IllegalArgumentException` . A boa prática diz que devemos preferir usar as já existentes do Java sempre que possível.

6. (opcional) Coloque um construtor na classe `SaldoInsuficienteException` que receba o valor que ele tentou sacar (isto é, ele vai receber um `double valor`).

Quando estendemos uma classe, não herdamos seus construtores, mas podemos acessá-los através da palavra chave `super` de dentro de um construtor. As exceções do Java possuem uma série de construtores úteis para poder populá-las já com uma mensagem de erro. Então vamos criar um construtor em `SaldoInsuficienteException` que delegue para o construtor de sua mãe. Essa vai guardar essa mensagem para poder mostrá-la ao ser invocado o método `getMessage` :

```java
public class SaldoInsuficienteException extends RuntimeException {

    public SaldoInsuficienteException(double valor) {
        super("Saldo insuficiente para sacar o valor de: " + valor);
    }
}
```

Dessa maneira, na hora de dar o `throw new SaldoInsuficienteException` você vai precisar passar

```
esse valor como argumento:
```

```java
if (this.saldo < valor) {
    throw new SaldoInsuficienteException(valor);
}
```

Atenção: você pode se aproveitar do Eclipse para isso: comece já passando o `valor` como

```
argumento para o construtor da exception e o Eclipse vai reclamar que não existe tal construtor.
O quick fix (`ctrl + 1`) vai sugerir que ele seja construindo, poupando-lhe tempo!
```

E agora, como fica o método `saca` da classe `ContaCorrente` ?

Resposta:

```java
@Override
public void saca(double valor) {
    if (valor < 0) {
        throw new IllegalArgumentException("Valor menor do que 0");
    }
    if (this.saldo < valor) {
        throw new SaldoInsuficienteException(valor);
    }
    this.saldo -= (valor + 0.10);
}
```

7. (opcional) Declare a classe `SaldoInsuficienteException` como filha direta de `Exception` em vez de `RuntimeException` . Ela passa a ser **checked**. O que isso resulta?

   Você vai precisar avisar que o seu método `saca()` `throws SaldoInsuficienteException` , pois ela é uma *checked* exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre `try-catch` ou `throws` . Faça uso do quick fix do Eclipse novamente!

   Depois, retorne a exception para *unchecked*, isto é, para ser filha de `RuntimeException` , pois utilizaremos ela assim em exercícios dos capítulos posteriores.

   Resposta: A mudança na classe `SaldoInsuficienteException` é apenas na classe mãe:

```java
public class SaldoInsuficienteException extends Exception {
    //...
}
```

   E, por conta disso, o método `saca` da classe `ContaCorrente` precisa avisar que pode, eventualmente, lançar essa exceção:

```java
public void saca(double valor) throws SaldoInsuficienteException {
    if (valor < 0) {
        throw new IllegalArgumentException();
    }
    if (this.saldo < valor) {
        throw new SaldoInsuficienteException(valor);
    }
    this.saldo -= (valor + 0.10);
}
```

23.13 DESAFIOS 12.12

1. O que acontece se acabar a memória da java virtual machine?

   Resposta: O que acontece é um `java.lang.OutOfMemoryError` , que **é um** `Error` em vez de uma `Exception` . http://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html

   O código para fazer esse erro é:

   ```java
   public class TestaError {
       public static void main(String[] args) {
           String[] ss = new String[Integer.MAX_VALUE];
       }
   }
   ```


23.14 EXERCÍCIOS 13.5: JAVA.LANG.OBJECT

1. Como verificar se a classe `Throwable` que é a superclasse de `Exception` também reescreve o método `toString` ?

   Resposta: A maioria das classes do Java que são muito utilizadas terão seus métodos `equals` e `toString` reescritos convenientemente.

Há algumas formas de verificar a sobrescrita de um método:

- Olhar o JavaDoc: se o método estiver sobrescrito, seu novo comportamento estará documentado alí;
- Abrir a classe e olhar: no Eclipse, se você tiver adicionado o *src.zip* nas suas configurações, pode abrir a classe com **ctrl + shift + T** e olhar se o método foi sobrescrito;
- Bom e velho Syso: outra possibilidade é criar objetos iguais, comparar com o `equals` e ver se funciona. Os outros métodos são, no entanto, bem mais eficientes.

2. Utilize-se da documentação do Java e descubra de que classe é o objeto referenciado pelo atributo `out` da `System`.

   Repare que, com o devido `import`, poderíamos escrever:

```
// falta a declaração da saída
________ saida = System.out;
saida.println("ola");
```

   A variável `saida` precisa ser declarada de que tipo? É isso que você precisa descobrir. Se você digitar esse código no Eclipse, ele vai te sugerir um quickfix e declarará a variável para você.

   Estudaremos essa classe em um capítulo futuro.

   Resposta: A variável pública e estática `out` é do tipo `PrintStream`.

3. Rode a aplicação e cadastre duas contas. Na tela de detalhes de conta, verifique o que aparece na caixa de seleção de conta para transferência. Por que isso acontece?

   Resposta: Nesse primeiro momento, algo parecido com isso deve ser mostrado:

```
br.com.caelum.contas.modelo.ContaCorrente@f34a08
```

   Porque o `toString` ainda não foi sobrescrito.

4. Reescreva o método `toString` da sua classe `Conta` fazendo com que uma mensagem mais explicativa seja devolvida. Lembre-se de aproveitar dos recursos do Eclipse para isto: digitando apenas o começo do nome do método a ser reescrito e pressionando **ctrl + espaço**, ele vai sugerir reescrever o método, poupando o trabalho de escrever a assinatura do método e cometer algum engano.

```
public abstract class Conta {

  protected double saldo;

    @Override
    public String toString() {
        return "[titular=" + titular + ", numero=" + numero
            + ", agencia=" + agencia + "]";
    }
  // restante da classe
```

```
}
```

Rode a aplicação novamente, cadastre duas contas e verifique novamente a caixa de seleção da transferência. O que aconteceu?

Resposta: Dessa vez, o resultado foi mais agradável. Deve ter aparecido algo como:

```
[titular=Batman, numero=123, agencia=345]
```

5. Reescreva o método `equals` da classe `Conta` para que duas contas com o mesmo **número e agência** sejam consideradas iguais. Esboço:

```
public abstract class Conta {

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }

        Conta outraConta = (Conta) obj;

        return this.numero == outraConta.numero &&
            this.agencia.equals(outraConta.agencia);
    }
}
```

Você pode usar o **ctrl + espaço** do Eclipse para escrever o esqueleto do

```
método `equals`, basta digitar dentro da classe `equ` e pressionar
**ctrl + espaço**.
```

Rode a aplicação e tente adicionar duas contas com o mesmo número e agência. O que acontece?

23.15 EXERCÍCIOS 13.7: JAVA.LANG.STRING

1. Queremos que as contas apresentadas na caixa de seleção da transferência apareçam com o nome do titular em maiúsculas. Para fazer isso vamos alterar o método `toString` da classe `Conta`. Utilize o método `toUpperCase` da `String` para isso.

Resposta:

```
@Override
public String toString() {
      return "[titular=" + titular.toUpperCase() + ", numero="
        + numero + ", agencia=" + agencia + "]";
  }
```

2. Após alterarmos o método `toString`, aconteceu alguma mudança com o nome do titular que é apresentado na lista de contas? Por que?

Resposta: Não mudou nada pois os métodos da `String` sempre retornam uma nova `String` mantendo o titular da conta inalterado.

3. Teste os exemplos desse capítulo, para ver que uma `String` é imutável.

   Resposta: Exemplos de teste:

```
public class TestaString {

    public static void main(String[] args) {
        String s = "fj11";
        s.replaceAll("1", "2");
        System.out.println(s);
    }

}
```

   Como fazer para ele imprimir fj22? *Resposta*:

```
public class TestaString {
    public static void main(String[] args) {
        String s = "fj11";
        String outra = s.replaceAll("1", "2");
        System.out.println(s);
        System.out.println(outra);
    }
}
```

4. Como fazer para saber se uma `String` se encontra dentro de outra? E para tirar os espaços em branco das pontas de uma `String` ? E para saber se uma `String` está vazia? E para saber quantos caracteres tem uma `String` ?

   Tome como hábito sempre pesquisar o JavaDoc! Conhecer a API, aos poucos, é fundamental para que você não precise reescrever a roda!

   Abra a página da documentação da classe String da versão do Java que você utiliza. http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

   Resposta: Os exemplos dessa questão são:

   - `contains` : devolve true se a `String` contem a sequência de caracteres passada;
   - `trim` : devolve uma nova `String` sem caracteres brancos do início e do fim;
   - `isEmpty` : devolve true se a `String` está vazia. Surgiu no Java 6;
   - `length` : devolve a quantidade de caracteres da `String` .

5. (opcional) Escreva um método que usa os métodos `charAt` e `length` de uma `String` para imprimir a mesma caractere a caractere, com cada caractere em uma linha diferente.

   Resposta:

```
    public void imprimeLetraPorLetra(String texto) {
        for (int i = 0; i < texto.length(); i++) {
            System.out.println(texto.charAt(i));
        }
    }
```

6. (opcional) Reescreva o método do exercício anterior, mas modificando ele para que imprima a `String` de trás para a frente e em uma linha só. Teste-a para *"Socorram-me, subi no ônibus em Marrocos"* e *"anotaram a data da maratona"*.

   Resposta:

```
public void inverte(String texto) {
    for (int i = texto.length() - 1; i >= 0; i--) {
        System.out.print(texto.charAt(i));
    }
    System.out.println("");
}
```

7. (opcional) Pesquise a classe `StringBuilder` (ou StringBuffer no Java 1.4). Ela é mutável. Por que usá-la em vez da `String`? Quando usá-la?

   Como você poderia reescrever o método de escrever a `String` de trás para a frente usando um `StringBuilder`? *Resposta*:

```
public void inverteComStringBuilder(String texto) {
    System.out.print("\t");
    StringBuilder invertido = new StringBuilder(texto).reverse();
    System.out.println(invertido);
}
```

23.16 DESAFIO 13.8

1. Converta uma `String` para um número sem usar as bibliotecas do java que já fazem isso. Isso é, uma `String x = "762"` deve gerar um `int i = 762`.

   Para ajudar, saiba que um `char` pode ser "transformado" em `int` com o mesmo valor numérico fazendo:

```
char c = '3';
int i = c - '0'; // i vale 3!
```

   Aqui estamos nos aproveitando do conhecimento da tabela unicode: os números de 0 a 9 estão em sequência! Você poderia usar o método estático `Character.getNumericValue(char)` em vez disso.

   Resposta:

```
public class DesafioConversaoDeNumeros {

    public static void main(String[] args) {
        String numero = "762";
        System.out.println("Imprimindo a string: " + numero);

        int resultado = converteParaInt(numero);
        System.out.println("Imprimindo o int: " + resultado);
    }
```

```
        private static int converteParaInt(String numero) {
            int resultado = 0;
            while (numero.length() > 0) {
                char algarismo = numero.charAt(0);
                resultado = resultado * 10 + (algarismo - '0');
                numero = numero.substring(1);
            }
            return resultado;
        }
    }
```


23.17 EXERCÍCIOS 14.5: ARRAYS

Para consolidarmos os conceitos sobre arrays, vamos fazer alguns exercícios que não interferem em nosso projeto.

1. Crie uma classe `TestaArrays` e no método `main` crie um array de contas de tamanho 10. Em seguida, faça um laço para criar 10 contas com saldos distintos e colocá-las no array. Por exemplo, você pode utilizar o índice do laço e multiplicá-lo por 100 para gerar o saldo de cada conta:

   ```
   Conta[] contas = new Conta[10];

   for (int i = 0; i < contas.length; i++) {
     Conta conta = new ContaCorrente();
     conta.deposita(i * 100.0);
     // escreva o código para guardar a conta na posição i do array
   }
   ```

   A seguir a resposta completa para este item:

   ```
   public class TestaArrays {
       public static void main(String[] args) {
       Conta[] contas = new Conta[10];

       for (int i = 0; i < contas.length; i++) {
           Conta conta = new ContaCorrente();
           conta.deposita(i * 100.0);
   ```

```
            contas[i] = conta;
        }
        }
}
```

2. Ainda na classe `TestaArrays` , faça um outro laço para calcular e imprimir a média dos saldos de todas as contas do array.

   Resposta:

```
public class TestaArrays {
    public static void main(String[] args) {
    Conta[] contas = new Conta[10];

    for (int i = 0; i < contas.length; i++) {
        Conta conta = new ContaCorrente();
        conta.deposita(i * 100.0);
        contas[i] = conta;
    }

    double soma = 0.0;
    for (int i = 0; i < contas.length; i++) {
        soma += contas[i].getSaldo();
    }
    double media = soma / contas.length;
    System.out.println("A média dos saldos é: " + media);
    }
}
```

3. (opcional) Crie uma classe `TestaSplit` que reescreva uma frase com as palavras na ordem invertida. *"Socorram-me, subi no ônibus em Marrocos"* deve retornar *"Marrocos em ônibus no subi Socorram-me,"*. Utilize o método `split` da `String` para te auxiliar. Esse método divide uma `String` de acordo com o separador especificado e devolve as partes em um array de `String` , por exemplo:

```
    String frase = "Uma mensagem qualquer";
    String[] palavras = frase.split(" ");

    // Agora só basta percorrer o array na ordem inversa imprimindo as palavras
```

   A seguir a resposta completa para este item:

```
public void invertePalavrasDaFrase(String texto) {
    String[] palavras = texto.split(" ");
    for (int i = palavras.length - 1; i >= 0; i--) {
    System.out.print(palavras[i] + " ");
    }
    System.out.println("");
}
```

4. (opcional) Crie uma classe `Banco` dentro do pacote `br.com.caelum.contas.modelo` O `Banco` deve ter um nome e um número (obrigatoriamente) e uma referência a uma array de `Conta` de tamanho 10, além de outros atributos que você julgar necessário.

   Resposta:

```
public class Banco {
private String nome;
private int numero;
private Conta[] contas;

// outros atributos que você achar necessário

public Banco(String nome, int numero) {
    this.nome = nome;
    this.numero = numero;
    this.contas = new ContaCorrente[10];
}

// getters para nome e número, não colocar os setters pois já recebemos no
// construtor
}
```

5. (opcional) A classe `Banco` deve ter um método `adiciona`, que recebe uma referência a `Conta` como argumento e guarda essa conta.

   Resposta: Você deve inserir a `Conta` em uma posição da array que esteja livre. Existem várias maneiras para você fazer isso: guardar um contador para indicar qual a próxima posição vazia ou procurar por uma posição vazia toda vez. O que seria mais interessante?

   Se quiser verificar qual a primeira posição vazia (nula) e adicionar nela, poderia ser feito algo como:

```
public void adiciona(Conta c) {
    for(int i = 0; i < this.contas.length; i++){
    // verificar se a posição está vazia
    // adicionar no array
    }
}
```

   É importante reparar que o método adiciona não recebe titular, agencia, saldo, etc. Essa seria uma maneira nem um pouco estruturada, muito menos orientada a objetos de se trabalhar. Você antes cria uma `Conta` e já passa a referência dela, que dentro do objeto possui titular, saldo, etc.

   Resposta:

```
public void adiciona(Conta c) {
    for(int i = 0; i < this.contas.length; i++){
        if(this.contas[i] == null) {
            this.contas[i] = c;
            break;
        }
    }
}
```

6. (opcional) Crie uma classe `TestaBanco` que possuirá um método `main`. Dentro dele crie algumas instâncias de `Conta` e passe para o banco pelo método `adiciona`.

   Resposta:

```
Banco banco = new Banco("CaelumBank", 999);
//     ....
```

Crie algumas contas e passe como argumento para o `adiciona` do banco:

Resposta:

```
ContaCorrente c1 = new ContaCorrente();
c1.setTitular("Batman");
c1.setNumero(1);
c1.setAgencia(1000);
c1.deposita(100000);
banco.adiciona(c1);

ContaPoupanca c2 = new ContaPoupanca();
c2.setTitular("Coringa");
c2.setNumero(2);
c2.setAgencia(1000);
c2.deposita(890000);
banco.adiciona(c2);
```

Você pode criar essas contas dentro de um loop e dar a cada um deles valores diferentes de depósitos:

```
for (int i = 0; i < 5; i++) {
    ContaCorrente conta = new ContaCorrente();
    conta.setTitular("Titular " + i);
    conta.setNumero(i);
    conta.setAgencia(1000);
    conta.deposita(i * 1000);
    banco.adiciona(conta);
}
```

Repare que temos de instanciar `ContaCorrente` dentro do laço. Se a instanciação de `ContaCorrente` ficasse acima do laço, estaríamos adicionado cinco vezes a **mesma** instância de `ContaCorrente` neste `Banco` e apenas mudando seu depósito a cada iteração, que nesse caso não é o efeito desejado.

Opcional: o método `adiciona` pode gerar uma mensagem de erro indicando quando o array já está cheio.

```
public class TestaBanco {

    public static void main (String[] args) {
        Banco banco = new Banco("CaelumBank", 999);

        ContaCorrente c1 = new ContaCorrente();
        c1.setTitular("Batman");
        c1.setNumero(1);
        c1.setAgencia(1000);
        c1.deposita(100000);
        banco.adiciona(c1);

        ContaPoupanca c2 = new ContaPoupanca();
        c2.setTitular("Coringa");
        c2.setNumero(2);
        c2.setAgencia(1000);
        c2.deposita(890000);
        banco.adiciona(c2);
    }
```

```
}
```

7. (opcional) Percorra o atributo `contas` da sua instância de `Banco` e imprima os dados de todas as suas contas. Para fazer isso, você pode criar um método chamado `mostraContas` dentro da classe `Banco`:

```
public void mostraContas() {
    for (int i = 0; i < this.contas.length; i++) {
        System.out.println("Conta na posição " + i);
        // preencher para mostrar outras informacoes da conta
    }
}
```

   Cuidado ao preencher esse método: alguns índices do seu array podem não conter referência para uma `Conta` construída, isto é, ainda se referirem para `null`. Se preferir, use o `for` novo do java 5.0.

   Aí, através do seu `main`, depois de adicionar algumas contas, basta fazer:

```
    banco.mostraContas();

    public void mostraContas() {
        for (int i = 0; i < this.contas.length; i++) {
            Conta conta = this.contas[i];
            if (conta != null) {
                System.out.println("Conta na posição: " + i);
                System.out.println("Saldo da conta: " + conta.getSaldo());
            }
        }
    }
```

   E, também, não esqueça de alterar a classe `TestaBanco`:

```
    public class TestaBanco {

        public static void main (String[] args) {
            // criação das contas...

            banco.mostraContas();
        }
    }
```

8. (opcional) Em vez de mostrar apenas o salário de cada funcionário, você pode usar o método `toString()` de cada `Conta` do seu array.

   Resposta:

```
    public void mostraContas() {
    for (int i = 0; i < this.contas.length; i++) {
        Conta conta = this.contas[i];
        if (conta != null) {
            System.out.println("Conta na posição: " + i);
            System.out.println("Dados da conta: " + conta);
        }
    }
}
```

9. (opcional) Crie um método para verificar se uma determinada `Conta` se encontra ou não como conta deste banco:

```
public boolean contem(Conta conta) {
// ...
}
```

Você vai precisar fazer um `for` em seu array e verificar se a conta passada como argumento se encontra dentro do array. Evite ao máximo usar números hard-coded, isto é, use o `.length`.

```
public boolean contem(Conta conta) {
    for (int i = 0; i < this.contas.length; i++) {
        if (contas.equals(this.contas[i])) {
            return true;
        }
    }
    return false;
}
```

10. (opcional) Caso o array já esteja cheio no momento de adicionar uma outra conta, crie um array novo com uma capacidade maior e copie os valores do array atual. Isto é, vamos fazer a realocação dos elementos do array já que java não tem isso: um array nasce e morre com o mesmo length.

Usando o this para passar argumento

Dentro de um método, você pode usar a palavra `this` para referenciar a si mesmo e pode passar essa referência como argumento.

```
public class Banco {

    // atributos

    public void adiciona(Conta c) {
        for(int i = 0; i < this.contas.length; i++){
            if(this.contas[i] == null) {
                this.contas[i] = c;
                return;
            }
        }
        this.aumentaArray();
    }

    public void aumentaArray() {
        int novoTamanho = this.contas.length * 2;
        Conta[] maior = new Conta[novoTamanho];
        for (int i = 0; i < this.contas.length; i++) {
            maior[i] = this.contas[i];
        }
        this.contas = maior;
    }

    // outros métodos
}
```

23.18 EXERCÍCIOS 15.6: ORDENAÇÃO

Vamos ordenar o campo de **destino** da tela de detalhes da conta para que as contas apareçam em ordem alfabética de titular.

1. Faça sua classe `Conta` implementar a interface `Comparable<Conta>`. Utilize o critério de ordenar pelo titular da conta.

```
public class Conta implements Comparable<Conta> {
    ...
}
```

Resposta: Deixe o seu método `compareTo` parecido com este:

```
public class Conta implements Comparable<Conta> {

    // ... todo o código anterior fica aqui

    public int compareTo(Conta outraConta) {
            return this.titular.compareTo(outraConta.titular);
    }
}
```

2. Queremos que as contas apareçam no campo de destino ordenadas pelo titular. Vamos então criar o método `ordenaLista` na classe `ManipuladorDeContas`. Use o `Collections.sort()` para ordenar a lista recuperada do `Evento`:

Resposta:

```
public class ManipuladorDeContas {

    // outros métodos

    public void ordenaLista(Evento evento) {
        List<Conta> contas = evento.getLista("destino");
        Collections.sort(contas);
    }
}
```

Rode a aplicação, adicione algumas contas e verifique se as contas aparecem ordenadas pelo nome do titular, **no campo destino**, na parte da transferência. Para ver a ordenação é necessário acessar os detalhes de uma conta.

Atenção especial: repare que escrevemos um método `compareTo` em nossa classe e nosso código **nunca** o invoca!! Isto é muito comum. Reescrevemos (ou implementamos) um método e quem o invocará será um outro conjunto de classes (nesse caso, quem está chamando o `compareTo` é o `Collections.sort`, que o usa como base para o algoritmo de ordenação). Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe `Collections` nunca imaginou que ordenaria objetos do tipo `Conta`, mas já que eles são `Comparable`, o seu método `sort` está satisfeito.

3. O que teria acontecido se a classe `Conta` não implementasse `Comparable<Conta>` mas tivesse o

método `compareTo` ?

Faça um teste: remova temporariamente a sentença `implements Comparable<Conta>` , não remova o método `compareTo` e veja o que acontece. Basta ter o método, sem assinar a interface?

Resposta: Não basta! A interface é como um contrato e sem "assiná-lo" a existência do método é só uma coincidência e não dá a certeza para a JVM de que a intenção era mesmo assinar aquele contrato.

4. Como posso inverter a ordem de uma lista? Como posso embaralhar todos os elementos de uma lista? Como posso rotacionar os elementos de uma lista?

   Investigue a documentação da classe `Collections` dentro do pacote `java.util` .

   Resposta: Olhando na documentação da classe `Collections` (http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html), encontramos o método `reverse()` , que recebe uma `List` e altera a ordem dos seus elementos, invertendo-os.

5. (opcional) Crie uma nova classe `TestaLista` que cria uma `ArrayList` e insere novas contas com saldos aleatórios usando um laço (`for`). Adivinhe o nome da classe para colocar saldos aleatórios? `Random` . Do pacote `java.util` . Consulte sua documentação para usá-la (utilize o método `nextInt()` passando o número máximo a ser sorteado).

   Resposta:

```
public class TestaLista {

    public static void main(String[] args) {
        List<Conta> contas = new ArrayList<Conta>();
        Random random = new Random();

        ContaPoupanca c1 = new ContaPoupanca(random.nextInt(2000), "Caio");
        c1.deposita(random.nextInt(10000) + random.nextDouble());
        contas.add(c1);

        ContaPoupanca c2 = new ContaPoupanca(random.nextInt(2000), "Adriano");
        c2.deposita(random.nextInt(10000) + random.nextDouble());
        contas.add(c2);

        ContaPoupanca c3 = new ContaPoupanca(random.nextInt(2000), "Victor");
        c3.deposita(random.nextInt(10000) + random.nextDouble());
        contas.add(c3);
    }
}
```

6. Modifique a classe `TestaLista` para utilizar uma `LinkedList` em vez de `ArrayList` :

```
    List<Conta> contas = new LinkedList<Conta>();
```

   Precisamos alterar mais algum código para que essa substituição funcione? Rode o programa. Alguma diferença?

Resposta: Essa mudança simplesmente funciona! O legal de chamar as coleções pelas suas interfaces é isso: não importa a implementação. Como ambas **são uma** `List` , é possível trocar entre elas e continuar tratando como `List` .

É mais uma aplicação do **polimorfismo**!

7. (opcional) Imprima a referência para essa lista. O `toString` de uma `ArrayList` / `LinkedList` é reescrito?

```
System.out.println(contas);
```

Resposta: Sim! Ele mostra os elementos da lista, entre colchetes e separados por vírgulas.

23.19 EXERCÍCIOS 15.15: COLLECTIONS

1. Crie um código que insira 30 mil números numa `ArrayList` e pesquise-os. Vamos usar um método de `System` para cronometrar o tempo gasto:

```
public class TestaPerformance {

    public static void main(String[] args) {
        System.out.println("Iniciando...");
        Collection<Integer> teste = new ArrayList<>();
        long inicio = System.currentTimeMillis();

        int total = 30000;

        for (int i = 0; i < total; i++) {
            teste.add(i);
        }

        for (int i = 0; i < total; i++) {
            teste.contains(i);
        }

        long fim = System.currentTimeMillis();
        long tempo = fim - inicio;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Troque a `ArrayList` por um `HashSet` e verifique o tempo que vai demorar:

```
Collection<Integer> teste = new HashSet<>();
```

O que é mais lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada `for` separadamente.

A diferença é mais que gritante. Se você passar de 30 mil para um número maior, como 50 ou 100 mil, verá que isso inviabiliza por completo o uso de uma `List` , no caso em que queremos utilizá-la essencialmente em pesquisas.

Resposta: No caso das listas (`ArrayList` e `LinkedList`), a inserção é bem rápida e a busca **muito lenta**!

No caso dos conjuntos (`TreeSet` e `HashSet`), a inserção ainda é rápida, embora um pouco mais lenta do que a das listas. E a busca é **muito rápida**!

2. (conceitual, importante) Repare que, se você declarar a coleção e der `new` assim:

```
Collection<Integer> teste = new ArrayList<>();
```

em vez de:

```
ArrayList<Integer> teste = new ArrayList<>();
```

É garantido que vai ter de alterar só essa linha para substituir a implementação por `HashSet` . Estamos aqui usando o polimorfismo para nos proteger que mudanças de implementação venham nos obrigar a alterar muito código. Mais uma vez: *programe voltado a interface, e não à implementação*!

Resposta: Esse é um **excelente** exemplo de bom uso de interfaces, afinal, de que importa como a coleção funciona? O que queremos é uma coleção qualquer, isso é suficiente para os nossos propósitos! Nosso código está com **baixo acoplamento** em relação a estrutura de dados utilizada: podemos trocá-la facilmente.

Esse é um código extremamente elegante e flexível. Com o tempo você vai reparar que as pessoas tentam programar sempre se referindo a essas interfaces menos específicas, na medida do possível: métodos costumam receber e devolver `Collection` s, `List` s e `Set` s em vez de referenciar diretamente uma implementação. É o mesmo que ocorre com o uso de `InputStream` e `OutputStream` : eles são o suficiente, não há porque forçar o uso de algo mais específico.

Obviamente, algumas vezes não conseguimos trabalhar dessa forma e precisamos usar uma interface mais específica ou mesmo nos referir ao objeto pela sua implementação para poder chamar alguns métodos. Por exemplo, `TreeSet` tem mais métodos que os definidos em `Set` , assim como `LinkedList` em relação à `List` .

Dê um exemplo de um caso em que não poderíamos nos referir a uma coleção de elementos como `Collection` , mas necessariamente por interfaces mais específicas como `List` ou `Set` .

Quando precisamos colocar a semântica de que uma coleção não pode ter repetição, por exemplo, precisamos de um `Set` . Se precisamos necessariamente de ordem, precisamos de uma `List` .

Pense na preparação de um mochilão pela Europa. Se eu estou interessado em contar para meus amigos por quais países eu vou passar, a repetição não importa, então eu escolheria um `Set` .

Agora, se eu quero planejar as passagens de um local a outro dessa viagem, não só a repetição de

locais importa como, também, a ordem. Então, preciso de uma `List` .

3. Faça testes com o `Map` , como visto nesse capítulo:

```
public class TestaMapa {

    public static void main(String[] args) {
        Conta c1 = new ContaCorrente();
        c1.deposita(10000);

        Conta c2 = new ContaCorrente();
        c2.deposita(3000);

        // cria o mapa
        Map mapaDeContas = new HashMap();

        // adiciona duas chaves e seus valores
        mapaDeContas.put("diretor", c1);
        mapaDeContas.put("gerente", c2);

        // qual a conta do diretor?
        Conta contaDoDiretor = (Conta) mapaDeContas.get("diretor");
        System.out.println(contaDoDiretor.getSaldo());
    }
}
```

Depois, altere o código para usar o *generics* e não haver a necessidade do casting, além da garantia de que nosso mapa estará seguro em relação a tipagem usada.

Você pode utilizar o *quickfix* do Eclipse para que ele conserte isso para você: na linha em que você está chamando o `put` , use o `ctrl + 1` . Depois de mais um quickfix (descubra!) seu código deve ficar como segue:

```
// cria o mapa
Map<String, Conta> mapaDeContas = new HashMap<>();
```

Que opção do `ctrl + 1` você escolheu para que ele adicionasse o *generics* para você?

Resposta: Há duas opções válidas aqui:

- podemos usar o *Add type arguments to Map* e, depois, novamente *Add type arguments to HashMap*;
- podemos escolher direto a opção *Infer generic type arguments*, que já fará tudo com apenas um comando.

4. (opcional) Assim como no exercício 1, crie uma comparação entre `ArrayList` e `LinkedList` , para ver qual é a mais rápida para se adicionar elementos na primeira posição (`list.add(0, elemento)`), como por exemplo:

```
public class TestaPerformanceDeAdicionarNaPrimeiraPosicao {
    public static void main(String[] args) {
        System.out.println("Iniciando...");
        long inicio = System.currentTimeMillis();
```

```
        // trocar depois por ArrayList
        List<Integer> teste = new LinkedList<>();

        for (int i = 0; i < 30000; i++) {
            teste.add(0, i);
        }

        long fim = System.currentTimeMillis();
        double tempo = (fim - inicio) / 1000.0;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Seguindo o mesmo raciocínio, você pode ver qual é a mais rápida para se percorrer usando o `get(indice)` (sabemos que o correto seria utilizar o *enhanced for* ou o `Iterator`). Para isso, insira 30 mil elementos e depois percorra-os usando cada implementação de `List` .

Perceba que aqui o nosso intuito não é que você aprenda qual é o mais rápido, o importante é perceber que podemos tirar proveito do polimorfismo para nos comprometer apenas com a interface. Depois, quando necessário, podemos trocar e escolher uma implementação mais adequada as nossas necessidades.

Qual das duas listas foi mais rápida para adicionar elementos à primeira posição?

Resposta: A `LinkedList` é bem mais rápida para fazer a inserção **na primeira posição** do que a `ArrayList` . Isso é uma característica dos algoritmos dessas listas e estudada sob o tópico de *Análise de algoritmos* na literatura.

Você pode aprender mais sobre isso na apostila aberta do **CS-14**, que está em: http://www.caelum.com.br/apostila-java-estrutura-dados/

5. (opcional) Crie a classe `Banco` (caso não tenha sido criada anteriormente) no pacote `br.com.caelum.contas.modelo` que possui uma `List` de `Conta` chamada `contas` . Repare que numa lista de `Conta` , você pode colocar tanto `ContaCorrente` quanto `ContaPoupanca` por causa do polimorfismo.

   Crie um método `void adiciona(Conta c)` , um método `Conta pega(int x)` e outro `int pegaQuantidadeDeContas()` . Basta usar a sua lista e delegar essas chamadas para os métodos e coleções que estudamos.

   Como ficou a classe `Banco` ?

   Resposta:

```
    public class Banco {
        private List<Conta> contas = new ArrayList<>();;

        public void adiciona(Conta conta) {
            contas.add(conta);
        }
```

```
        public Conta pega(int posicao) {
            return contas.get(posicao);
        }

        public int getQuantidadeDeContas() {
            return contas.size();
        }
    }
```

6. (opcional) No `Banco` , crie um método `Conta buscaPorTitular(String nome)` que procura por uma `Conta` cujo `titular` seja `equals` ao `nomeDoTitular` dado.

   Você pode implementar esse método com um `for` na sua lista de `Conta` , porém não tem uma performance eficiente.

   Adicionando um atributo privado do tipo `Map<String, Conta>` terá um impacto significativo. Toda vez que o método `adiciona(Conta c)` for invocado, você deve invocar `.put(c.getTitular(), c)` no seu mapa. Dessa maneira, quando alguém invocar o método `Conta buscaPorTitular(String nomeDoTitular)` , basta você fazer o `get` no seu mapa, passando `nomeDoTitular` como argumento!

   Note, apenas, que isso é só um exercício! Dessa forma você não poderá ter dois clientes com o mesmo nome nesse banco, o que sabemos que não é legal.

   Como ficaria sua classe `Banco` com esse `Map` ?

   Resposta:

```
    public class Banco {
        private List<Conta> contas = new ArrayList<>();
        private Map<String, Conta> indexadoPorNome = new HashMap<>();

        public void adiciona(Conta conta) {
            contas.add(conta);
            indexadoPorNome.put(conta.getTitular(), conta);
        }

        public Conta buscaPorTitular(String nomeDoTitular) {
            return indexadoPorNome.get(nomeDoTitular);
        }
    }
```

7. (opcional, avançado) Crie o método `hashCode` para a sua conta, de forma que ele respeite o `equals` de que duas contas são `equals` quando tem o mesmo número e agência. Felizmente para nós, o próprio Eclipse já vem com um criador de `equals` e `hashCode` que os faz de forma consistente.

   Na classe `Conta` , use o `ctrl + 3` e comece a escrever *hashCode* para achar a opção de gerá-los. Então, selecione os atributos `numero` e `agencia` e mande gerar o `hashCode` e o `equals` .

Como ficou o código gerado?

Resposta:

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((agencia == null) ? 0 : agencia.hashCode());
    result = prime * result + numero;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Conta other = (Conta) obj;
    if (agencia == null) {
        if (other.agencia != null)
            return false;
    } else if (!agencia.equals(other.agencia))
        return false;
    if (numero != other.numero)
        return false;
    return true;
}
```

8. (opcional, avançado) Crie uma classe de teste e verifique se sua classe `Conta` funciona agora corretamente em um `HashSet`, isto é, que ela não guarda contas com número e agência repetidos. Depois, remova o método `hashCode`. Continua funcionando?

   Resposta: Dominar o uso e o funcionamento do `hashCode` é fundamental para o bom programador.

   Sem o `hashCode`, o critério para definir o que são contas iguais e o que são contas diferentes se perde e, assim, o `HashSet` não consegue garantir a aparição única de uma conta.

   A classe para fazer essa verificação fica mais ou menos assim.

```
public class TestaHashSetDeConta {

    public static void main(String[] args) {
        HashSet<Conta> contas = new HashSet<>();

        ContaCorrente c1 = new ContaCorrente();
        c1.setNumero(1);
        c1.setAgencia(1000);
        c1.setTitular("Batman");

        ContaCorrente c2 = new ContaCorrente();
        c2.setNumero(1);
        c2.setAgencia(1000);
        c2.setTitular("Robin");
```

```
        ContaCorrente c3 = new ContaCorrente();
        c3.setNumero(2);
        c3.setAgencia(1000);
        c3.setTitular("Coringa");

        contas.add(c1);
        contas.add(c2);
        contas.add(c3);

        System.out.println("Total de contas no HashSet: " + contas.size());
    }
}
```

Você pode também fazer o curso data dessa apostila na Caelum



Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso data** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

23.20 DESAFIOS 15.16

1. Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `TreeSet`. Como ficou seu código?

   Resposta:

```
public class TestaTreeSetDecrescente {

    public static void main(String[] args) {
        TreeSet<Integer> conjunto = new TreeSet<>();
        for (int i = 1; i <= 1000; i++) {
            conjunto.add(i);
        }

        for (Integer i : conjunto.descendingSet()) {
            System.out.print(i + " ");
        }
    }
}
```

2. Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `ArrayList`. Como ficou seu código? *Resposta:*

```
public class TestaArrayListDecrescente {
```

```
        public static void main(String[] args) {
            List<Integer> lista = new ArrayList<>();
            for (int i = 1; i <= 1000; i++) {
                lista.add(i);
            }

            Collections.sort(lista);
            Collections.reverse(lista);

            for (Integer i : lista) {
                System.out.print(i + " ");
            }
        }
    }
```

23.21 EXERCÍCIOS 16.11: JAVA I/O

Vamos salvar as contas cadastradas em um arquivo para não precisar ficar adicionando as contas a todo momento.

1. Na classe `ManipuladorDeContas` , crie o método `salvaDados` que recebe um `Evento` de onde obteremos a lista de contas:

   Resposta:

   ```
   public void salvaDados(Evento evento){
   List<Conta> contas = evento.getLista("listaContas");
   // aqui salvaremos as contas em arquivo
   }
   ```

2. Para não colocarmos todo o código de gerenciamento de arquivos dentro da classe `ManipuladorDeContas` , vamos criar uma nova classe cuja responsabilidade será lidar com a escrita / leitura de arquivos. Crie a classe `RepositorioDeContas` dentro do pacote `br.com.caelum.contas` e declare o método `salva` que deverá receber a lista de contas a serem guardadas. Neste método você deve percorrer a lista de contas e salvá-las separando as informações de `tipo` , `numero` , `agencia` , `titular` e `saldo` com vírgulas. O código ficará parecido com:

   Resposta:

   ```
   public class RepositorioDeContas {

       public void salva(List<Conta> contas) {
           PrintStream stream = new PrintStream("contas.txt");
           for (Conta conta : contas) {
               stream.println(conta.getTipo() + "," + conta.getNumero() + ","
                   + conta.getAgencia() + "," + conta.getTitular() + ","
                   + conta.getSaldo());
           }
           stream.close();
       }
   }
   ```

   O compilador vai reclamar que você não está tratando algumas exceções (como

`java.io.FileNotFoundException`). Utilize o devido `try` / `catch` e relance a exceção como `RuntimeException` . Utilize o *quick fix* do Eclipse para facilitar (**ctrl + 1**).

Vale lembrar que deixar todas as exceptions passarem despercebidas não é uma boa prática! Você pode usar aqui, pois estamos focando apenas no aprendizado da utilização do `java.io` .

Quando trabalhamos com recursos que falam com a parte externa à nossa aplicação, é preciso que avisemos quando acabarmos de usar esses recursos. Por isso, é **importantíssimo** lembrar de fechar os canais com o exterior que abrimos utilizando o método `close` !

3. Voltando à classe `ManipuladorDeContas` , vamos completar o método `salvaDados` para que utilize a nossa nova classe `RepositorioDeContas` criada.

   Resposta:

```
public void salvaDados(Evento evento){
    List<Conta> contas = evento.getLista("listaContas");
    RepositorioDeContas repositorio = new RepositorioDeContas();
    repositorio.salva(contas);
}
```

   Rode sua aplicação, cadastre algumas contas e veja se aparece um arquivo chamado `contas.txt` dentro do diretório `src` de seu projeto. Talvez seja necessário dar um F5 nele para que o arquivo apareça.

4. (Opcional, Difícil) Vamos fazer com que além de salvar os dados em um arquivo, nossa aplicação também consiga carregar as informações das contas para já exibir na tela. Para que a aplicação funcione, é necessário que a nossa classe `ManipuladorDeContas` possua um método chamado `carregaDados` que devolva uma `List<Conta>` . Vamos fazer o mesmo que anteriormente e encapsular a lógica de carregamento dentro da classe `RepositorioDeContas` :

```
public List<Conta> carregaDados() {
    RepositorioDeContas repositorio = new RepositorioDeContas();
    return repositorio.carrega();
}
```

   Faça o código referente ao método `carrega` que devolve uma `List` dentro da classe `RepositorioDeContas` utilizando a classe `Scanner` . Para obter os valores de cada atributo você pode utilizar o método `split` da `String` . Lembre-se que os atributos das contas são carregados na seguinte ordem: `tipo` , `numero` , `agencia` , `titular` e `saldo` . Exemplo:

```
    String linha = scanner.nextLine();
    String[] valores = linha.split(",");
    String tipo = valores[0];
```

   Além disso, a conta deve ser instanciada de acordo com o conteúdo do `tipo` obtido. Também fique atento pois os dados lidos virão sempre lidos em forma de `String` e para alguns atributos será

necessário transformar o dado nos tipos primitivos correspondentes. Por exemplo:

```
String numeroTexto = valores[1];
int numero = Integer.parseInt(numeroTexto);
```

A seguir a resposta completa para este item:

```
public List<Conta> carrega() {
        List<Conta> contas = new ArrayList<>();
        try (Scanner scanner = new Scanner(new File("contas.txt"))) {
            while (scanner.hasNextLine()) {
                Conta conta;
                String linha = scanner.nextLine();
                String[] valores = linha.split(",");
                String tipo = valores[0];
                int numero = Integer.parseInt(valores[1]);
                String agencia = valores[2];
                String titular = valores[3];
                double saldo = Double.parseDouble(valores[4]);

                if (tipo.equals("Conta Corrente")) {
                    conta = new ContaCorrente(numero, agencia, titular, saldo);
                } else {
                    conta = new ContaPoupanca(numero, agencia, titular, saldo);
                }
                contas.add(conta);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Não tem arquivo ainda");
        }
        return contas;
    }
```

5. (opcional) A classe `Scanner` é muito poderosa! Consulte seu javadoc para saber sobre o `delimiter` e os outros métodos `next` .

6. (opcional) Crie uma classe `TestaInteger` e vamos fazer comparações com Integers dentro do `main` :

```
Integer x1 = new Integer(10);
Integer x2 = new Integer(10);

if (x1 == x2) {
    System.out.println("igual");
} else {
    System.out.println("diferente");
}
```

E se testarmos com o `equals` ? O que podemos concluir?

Resposta: A conclusão é aquela mesma do capítulo de Orientação a Objeto do curso. Não importa que todos as informações sejam exatamente iguais: quando usamos o `==` estamos comparando as **variáveis**, isto é, a referência para objetos.

Se demos `new` duas vezes, cada referência aponta para um objeto diferente e, portanto, não são iguais. Já o `equals` do `Integer` , que sobrescreve o do `Object` , compara o conteúdo dos objetos.

7. (opcional) Um `double` não está sendo suficiente para guardar a quantidade de casas necessárias em uma aplicação. Preciso guardar um número decimal muito grande! O que poderia usar?

   O `double` também tem problemas de precisão ao fazer contas, por causa de arredondamentos da aritmética de ponto flutuante definido pela IEEE 754:

   http://en.wikipedia.org/wiki/IEEE_754

   Ele não deve ser usado se você precisa realmente de muita precisão (casos que envolvam dinheiro, por exemplo).

   Consulte a documentação, tente adivinhar onde você pode encontrar um tipo que te ajudaria para resolver esses casos e veja como é intuitivo! Qual é a classe que resolveria esses problemas?

   Lembre-se: no Java há muito já pronto. Seja na biblioteca padrão, seja em bibliotecas *open source* que você pode encontrar pela internet.

   Resposta: A classe que nos ajudará a evitar arredondamentos e a armazenar números decimais bem grandes é a `java.math.BigDecimal`